MASTER EN SCIENCES INFORMATIQUES

De l'adéquation de la programmation logique aux systèmes à bases de connaissances : A.I.D.A.

Moisse, Philippe

Award date:
1990

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

" De l'adéquation de la programmation logique aux systèmes à bases de connaissances : A.I.D.A. "

Promoteur : Jean RAMAEKERS.

Mémoire présenté par Philippe MOISSE
en vue de l'obtention du titre de
licencié et maître en
informatique.

Année académique 1989-1990.

Remerciements.

Ce mémoire a été rédigé sous la direction de Monsieur le Professeur Jean Ramaekers, qui m'a également procuré le stage chez Siemens à Munich où l'application a été développée. Je le remercie sincèrement pour l'aide efficace qu'il m'a apportée tout au long de mon stage et pour le suivi constant qu'il a assuré lors de la réalisation de cet ouvrage. Je tiens également à remercier Monsieur Philippe Dumont, mon maître de stage pour Siemens Belgique qui m'a fourni une étude très poussée des fonctionnalités attendues de l'application et de quelques solutions qui pouvaient leur être appliquées. Monsieur Bernard Bolle, qui avait déjà travaillé sur le projet en suivant une autre approche, m'a donné l'impulsion initiale nécessaire; qu'il reçoive toute ma gratitude ainsi que toute l'équipe ST SP 113 à Munich, et plus particulièrement Messieurs Gärre et Bernard Pauwels.

Je n'oublie pas toute ma famille, qui a supporté mon humeur souvent maussade tout au long de la réalisation de cet ouvrage ainsi que Corine et Alain pour l'aide directe et le soutien moral qu'ils m'ont apportés.

Résumé.

Lorsque des problèmes surviennent pendant l'exécution d'un système d'exploitation, la seule solution pratique pour les résoudre est de faire une copie binaire intégrale, appelée 'dump', de tout le système et de l'analyser ensuite. AIDA est un outil qui permet, sur base d'une description formelle des structures de données du système d'exploitation, de parcourir un 'dump' à la recherche d'incohérences ou d'erreurs. A ce titre, AIDA peut également se révéler très utile lors du développement et de la mise au point du système d'exploitation.

Abstract.

When problems occur in a system while the operating system is running, the only useful way to solve these problems is to make a full binary copy of the whole running system and to analyze it later. Such a copy is called a dump. AIDA, for ' an Artificially Intelligent Dump Analyzer ', is a tool that makes possible the walking of a dump in search for errors or incoherences. AIDA takes his knowledge of the structures of the operating systems from a built in formal description of these structures. AIDA can be found very useful too while developing and maintaining the operating system.

Introduction.

Un système d'exploitation constitue un logiciel dont le volume requiert une maintenance très lourde. Comme pour un développement incrémental tel celui d'une nouvelle version, cette maintenance exige des connaissances sur le logiciel que l'on ne peut attendre de chaque personne impliquée dans le développement.

Un projet existe chez Siemens et a pour but de développer un programme réunissant toutes les structures de données d'un logiciel de taille quelconque qui soit capable d'être interrogé par un opérateur. Appliqué à une représentation figée d'un programme en exécution telle qu'un 'dump', cet outil permettrait entre autre de rechercher un type d'erreur d'exécution ayant trait aux structures de données.

Le travail qui nous a été demandé consistait à développer ce programme et la base de données correspondante. Un projet apparenté a déjà été développé par Monsieur Bolle, dans un mémoire présenté à l'Institut, ce dans une approche de structuration pure des données. Cette approche, peu aisée pour la réalisation d'une interface d'interrogation, demandait à être revue. C'est ainsi que nous avons choisi d'implémenter notre application, programme et bases de connaissances, en Prolog.

Si elle était suffisamment puissante, notre application pouvait faire l'objet d'un développement complémentaire et d'une utilisation effective par la société Siemens. Cependant, notre développement n'a pas dépassé le cadre d'un prototype.

L'exposé est principalement centré sur l'application réalisée et sur les choix à la fois stratégiques et de représentation qui ont été faits; il tente également de justifier un certain style et une certaine vision de la programmation : ceux de la logique. Il essaie en outre de montrer les nombreux avantages offerts par le simple choix de Prolog.

Dans ce même cadre, l'approche logique de la programmation a été quelque peu élargie à Prolog et sa présentation cherche à pister les richesses sémantiques particulières qu'elle peut apporter dans la représentation des connaissances.

Au premier chapitre, après une brève introduction sur les idées de base de la logique et leur application à la programmation, nous présenterons le concept de connaissance qui permet entre autre de représenter le concept de données et sa structuration. Nous verrons ensuite le langage Prolog dans un niveau de détail suffisant pour la compréhension de notre application et nous traiterons de son adéquation à la représentation de bases de données et de bases de connaissances.

L'application réalisée occupera tout le deuxième chapitre. Nous y exposerons les besoins, la représentation des structures de données et le programme lui-même. Après avoir développé les détails de l'implémentation, nous livrerons dans une quatrième partie du chapitre les limitations de notre prototype. Une attention particulière sera accordée à la recherche de ces limites et des conséquences qu'elles peuvent engendrer.

Le troisième chapitre traitera des outils existant dans la lignée de la programmation logique. Une première partie survolera les différentes logiques à sémantique enrichie ; l'autre présentera Prolog II, la deuxième version du langage Prolog.

L'articulation du mémoire se veut centrée à la fois sur l'application réalisée et sur les promesses de l'approche logique en programmation. Nos réflexions sur ce dernier thème seront présentées dans la conclusion.

Chapitre 1 : La logique et les systèmes à bases de connaissances.

1. La logique et la programmation logique.

1.1. La logique.

Nous ne présenterons ici que les éléments de base nécessaires à la compréhension d'une approche logique de la programmation. Cette programmation se voulant de plus introduite par le langage Prolog, nous n'aborderons pas les thèmes de la logique tels que le lambda-calcul qui reflètent une autre vision de l'application de la logique à la programmation.

1.1.1. Le calcul des propositions.

1.1.1.1. Introduction.

Les briques de base du calcul des propositions sont les valeurs 'vrai' et 'faux'. Ces valeurs sont des formules. Une variable est également une formule. La construction de formules plus complexes se fait grâce aux cinq connecteurs logiques 'et', 'ou', 'négation', 'implication' et 'équivalence', notés respectivement 'et', 'ou', '~', '=>' et '<=>' dans ce travail. A ce stade de la théorie, une formule peut être de la forme suivante : '((vrai et (~(faux ou (vrai => faux)))) => (faux <=> vrai))'.

1.1.1.2. La sémantique des formules.

La logique a défini une manière univoque de déterminer la sémantique des formules, c'est-à-dire la valeur totale d'une formule. Pour ce faire, elle utilise le principe dénotationnel, ou principe de transparence, disant que ' si une formule fait partie d'une autre formule, et si cette partie est remplacée par une autre formule ayant la même valeur, la formule totale aura toujours la même valeur ' . En plus de ce principe, la logique nécessite une table d'équivalence entre formules. Notons v la valeur 'vrai', f la valeur 'faux' et a une valeur quelconque. Les équivalences logiques sont résumées dans le tableau suivant :

formules vraies	formules fausses
v	f
~f	~v
v et v'	f et a
v ou a	a et f
a ou v	f ou f'
a => v	v => f
f => a	v <=> f
v <=> v'	f <=> v
f <=> f'	

(figure 1.1.)

Grâce à ces équivalences, il est possible de simplifier la formule que nous avons prise en exemple comme suit :

((v et (~ (f ou (v => f)))) => (f <=> v))
((v et (~ f ou f))) => f)
((v et (~f)) => f)
((v et v) => f)
(v => f)
f

Notre formule a donc pour valeur 'faux'.

1.1.1.3. Consistence et complétude.

Deux propriétés que peuvent avoir les systèmes formels[1] sont la consistence et la complétude. La complétude d'un système formel assure que toutes les formules de ce système sont soit vraies soit fausses par rapport à leur sémantique. La consistence d'un même système assure, toujours par rapport à sa sémantique, qu'aucune formule n'est à la fois vraie et fausse.

[1] Les systèmes formels sont aussi appelés systèmes de formules.

La logique que nous avons définie possède ces deux propriétés. Mais il est intéressant de noter que si la consistence est essentielle, la complétude contrarie souvent, ne permettant pas de réserver son jugement sur certaines affirmations.

Un système consistent et complet peut posséder trois sortes de formules. L'une sera une tautologie, c'est-à-dire une formule qui est toujours vraie quelle que soit la valeur de ses constituants. Une autre consistera en une formule satisfiable, c'est-à-dire une formule pouvant être vraie sous une certaine combinaison de valeurs pour ses constituants. Une troisième formera une contradiction, une formule toujours fausse quelles que soient les valeurs de ses constituants.

1.1.1.4. Les lois.

Les lois sont les tautologies découvertes pour la logique habituelle. Elles sont très nombreuses et il serait inutile de les citer ici. Leur intérêt réside dans leur utilisation possible en tant que règles de ré-écriture de formules en autres formules équivalentes.

Ces lois seront notamment utilisées pour la déduction de théorèmes. Un axiome est une assertion que l'on pose comme vraie, un théorème est une assertion que l'on trouve être vraie si on la déduit des axiomes existants en appliquant les lois. Ainsi, on peut construire un modèle qui est une classification d'assertions soit vraies soit fausses et dans laquelle tous les axiomes sont vrais.

La logique offre donc une base formelle pour la construction de modèles. L'aspect le plus intéressant de cette théorie de la logique est la possibilité d'automatiser ses mécanismes de preuve, permettant une programmation de ces dernières et donc offrant l'occasion d'en faire un langage de programmation. Dans cette optique, un programme et une base de connaissances sont des modèles.

1.1.2. Le calcul des prédicats.

1.1.2.1. Présentation générale

L'élément de base de la théorie du calcul des prédicats est le prédicat. Il consiste en un symbole de prédicat pouvant être suivi d'arguments, c'est-à-dire d'une liste de termes entre parenthèses et séparés par des virgules. Un tel prédicat est une formule; plusieurs formules peuvent être reliées entre elles avec des connecteurs logiques pour former de nouvelles formules. Jusqu'ici, il n'y a aucun élément supplémentaire au calcul propositionnel.

La différence est amenée par la possibilité, dans le calcul des prédicats, de préfixer une formule par un quantificateur. Il existe deux quantificateurs : le quantificateur universel appelé 'pour tout' et le quantificateur existentiel appelé 'il existe'. Les quantificateurs s'appliqueront uniquement à une variable faisant partie de la formule.

Deuxième différence alors : l'existence de constantes et de variables dans le calcul des prédicats. Une constante est le nom d'un objet spécifique alors qu'une variable est le nom d'un objet arbitraire mais appartenant à une classe spécifique d'objets appelée son type.

Bien entendu, il faut encore donner à une formule une interprétation, et cette notion d'interprétation doit au moins englober la même notion du calcul propositionnel. On peut donc trouver des formules qui seront vraies[2] ou fausses[3] dans toutes les interprétations. Des lois existent également dans le calcul des prédicats et le principe de déduction de théorèmes est identique.

---

[2] Ces formules seront alors appelées des tautologies.

[3] Ces formules seront toujours qualifiées d'inconsistantes. Elles seront aussi appelées, comme dans le calcul propositionnel, des contradictions.

1.1.2.2. Le raisonnement formel

La notion de conséquence logique autorise le raisonnement formel, c'est-à-dire un raisonnement dans lequel on ne tient compte que de la forme syntaxique des formules et non de leurs interprétations.

La logique comporte des règles d'inférence permettant de déduire des conséquences logiques d'autres formules logiques. Un exemple d'une telle règle est le modus ponens : ' de toute formule A et de toute formule de la forme A => B, on peut toujours dériver la formule B '. D'autres règles existent, telles ' A => B <=> ~A ou B ' et ' ~(A et B) <=> ~A ou ~B '.

Avec ces règles d'inférence, on peut essayer de prouver qu'une formule est une tautologie grâce au principe de réduction par l'absurde. Ce principe est le suivant : pour prouver la formule A, supposer que ~A est vrai et, en combinant ~A avec toutes les tautologies connues grâce aux règles d'inférence, en arriver à une formule résultante B. Si B est inconsistante, alors A est une tautologie.

1.1.2.3. Les clauses de Horn

Une clause est une formule ne contenant que des connecteurs 'ou'. Une clause de Horn est une clause contenant au plus un seul terme qui ne soit pas nié. De telles formules sont très intéressantes car on peut très facilement y appliquer le principe de résolution.

Ce principe se base sur la règle suivante : ' de ~A ou B et de A ou C, on peut toujours dériver B ou C '. Deux cas particuliers de cette règle sont très intéressants : ' de ~A ou B et de A, dériver B ' et ' de ~A et de A dériver l'inconsistance '.

Grâce à ces deux dernières règles, le lecteur peut voir que si on ajoute la clause ~A à un ensemble de clauses de Horn, et si on applique le principe de réduction par l'absurde, on peut obtenir une clause inconsistante et donc prouver que les clauses forment une tautologie. C'est ce mécanisme qui est appelé principe de résolution.

Cette méthode de preuve peut être étendue et englober les quantificateurs. Le lecteur pourra en trouver le détail dans [Kluzniak 85] au chapitre 2. Comme plusieurs théorèmes permettent de ramener tout prédicat à une forme clausale, l'extension de théorèmes d'automatisation de preuve du calcul des propositions est possible et permet l'automatisation des preuves du calcul des prédicats.

Ces clauses, permettant de représenter des propriétés d'objets, forment la base de la programmation logique lorsqu'elles sont couplées au mécanisme de résolution; ce dernier est un mécanisme efficacement automatisable de preuve. Cette idée de programmation logique est incarnée par le langage Prolog.

1.2. La programmation logique.

1.2.1. La représentation par la logique.

La représentation par la logique de connaissances consiste à encoder des faits que l'on pose être vrais. Ce ne serait évidemment utile que si on pouvait appliquer des mécanismes de déduction sur ces représentations. C'est justement cette possibilité qu'offre le calcul des prédicats : on peut définir des prédicats unaires[4] tels homme(socrate[5]) disant que Socrate est un homme et des prédicats n-aires tels adresse(numéro(W), rue(X), code postal(Y), localité(Z)) spécifiant les caractéristiques d'une adresse.

La sémantique de ces représentations consiste à associer un objet réel à chaque constante. Bien que pouvant varier avec le contexte, la signification d'une telle représentation doit toujours être unique.

Au-delà des faits que nous avons présentés avec 'homme' et 'adresse', des règles sont également représentables, et donnent alors accès à l'utilisation des connecteurs

---

[4] Un prédicat unaire est un prédicat ne comportant qu'un seul argument. Un prédicat n-aire comporte n arguments.

[5] Le nom 'socrate' est une constante, n'ayant pas d'arguments entre parenthèses. Mais 'homme' est le nom d'un prédicat unaire, tout comme 'rue'. L'expression 'rue(X)' est également un objet composé. La logique confond donc un prédicat et un objet, accréditant l'usage de représenter un objet par un prédicat. Dans certains cas, une constante sera également considérée comme un prédicat.

logiques; ces règles permettant ainsi, grâce aux mécanismes de preuve automatisés, des déductions sur base des règles et des faits.

Voici un exemple d'une telle règle : ville (X) => localité (X) , qui signifie qu'une localité ne peut être considérée comme telle que si elle est bien une ville, sa définition en tant que ville ayant été enregistrée dans une quelconque base de connaissances. Ainsi, une preuve n'acceptera une adresse que si sa localité a déjà été définie comme étant une ville.

Ces preuves peuvent se faire dans deux directions : elles partent des faits et essaient de prouver un but grâce à tous les éléments qu'elles ont à leur disposition ou elles partent du but à prouver et essaient de le réduire par des règles en des faits représentés comme toujours vrais. La première direction est appelée déduction en avant et la deuxième déduction en arrière; cette dernière direction est d'ailleurs celle utilisée par Prolog. Les preuves peuvent également être implémentées selon deux approches distinctes : soit l'approche en profondeur d'abord, voulant épuiser toutes les ramifications d'un même chemin de recherche avant de passer au suivant, soit l'approche en largeur d'abord, qui crée tous les chemins possibles et les explore tous ensemble dans le sens de leur profondeur. Prolog utilise l'approche en profondeur d'abord.

1.2.2. Principe de la programmation logique.

L'idée-clé de la programmation logique est de remplacer l'algorithmique et la programmation par une description logique d'un problème et par un mécanisme de preuve automatique permettant de déduire la réponse attendue. Ainsi, les hypothèses sur lesquelles le programme se base sont décrites explicitement alors que la séquence d'opérations à effectuer pour résoudre le problème reste implicite.

Cette idée vient d'une volonté constante chez les programmeurs en général : celle d'écrire plutôt des spécifications que des programmes [Amble 87] . Avec l'avènement de langages implémentant parfaitement l'idée de la programmation logique, les programmeurs seront alors en possession d'un outil de spécification de problèmes et de résolution automatique de ceux-ci.

Un autre principe de la programmation logique est le calcul symbolique. Il induit que les identificateurs du langage se représentent eux-même et ne sont pas un quelconque endroit où l'on va garder une valeur numérique. Sur cette base, la programmation logique sera purement déclarative, même si son implémentation dans un langage de programmation doit accepter encore des aspects procéduraux.

C'est là que le lecteur peut remarquer que même la programmation logique n'est en rien une panacée : même si une de ses implémentations permettait une programmation purement déclarative, le problème du 'que' et du 'comment' spécifier serait toujours présent, induisant alors l'adéquation ou l'inadéquation de l'approche logique au problème à résoudre.

2. Les systèmes à bases de connaissances.

2.1. L'ingéniérie de la connaissance.

La tâche première de l'ingéniérie de la connaissance est de créer des systèmes experts, basés sur des connaissances et capables de résoudre des problèmes, c'est-à-dire aptes à trouver et à appliquer des règles et des heuristiques[6] gouvernant le processus de réflexion de l'expert humain. Les systèmes à bases de connaissances diffèrent donc des programmes traditionnels parce qu'ils sont en fait des sous-produits de ces règles, obtenus sans l'utilisation d'une vraie programmation dans le sens habituel du terme.

Cette tâche nécessite bien entendu la description d'un moyen de transfert des connaissances des experts vers l'ordinateur. Cette préoccupation ne sera pas abordée dans ce travail. Nous renvoyons le lecteur à [Garrico 89] qui, à partir du chapitre 3, propose une méthodologie de construction de systèmes à bases de connaissances et dans celle-ci une méthodologie d'acquisition des connaissances. Dans notre application, l'acquisition des connaissances fut réalisée par l'auteur sans se référer à une méthodologie précise.

---

[6] Une heuristique est une méthode connue de résolution de problème. Elle représente en général la manière de conduire la recherche des solutions au problème posé.

2.2. Les systèmes à bases de connaissances.

Les systèmes à bases de connaissances peuvent être considérés comme la base de l'intelligence artificielle appliquée. Après avoir constaté qu'il n'était pas très constructif de vouloir modéliser entièrement l'intelligence humaine en se rendant compte que la modélisation des erreurs humaines devenait aussi importante que la modélisation des capacités humaines [Amble 87] , l'intelligence artificielle s'est tournée vers une approche lui assurant la possession des connaissances et non l'acquisition d'une quelconque possibilité générale de résoudre des problèmes. Cela a d'ailleurs fait naître les systèmes experts.

Ainsi, si l'informatique en général sert à manipuler des données, l'intelligence artificielle s'est donné pour but de les interpréter. La recherche de systèmes basés sur des connaissances et non sur des données s'inscrit aisément dans cette volonté.

Mais l'approche des systèmes à bases de connaissances révèle aussi des difficultés. En effet, même dans un domaine d'expertise très restreint, toutes les connaissances courantes doivent être supposées connues alors que pour un système informatique elles ne le sont absolument pas. La programmation de l'expertise étant en fait la tâche la plus ardue, on a préféré utiliser, pour écrire des systèmes experts, des systèmes à bases de connaissances possédant déjà tout un mécanisme de gestion de connaissances et ne nécessitant que la description de telles connaissances pour pouvoir travailler efficacement.

Il faut également noter que des systèmes à bases de connaissances ne possèdent en fait aucune intelligence. Ils ont une sémantique pour une représentation formelle des connaissances ainsi que des procédures de déduction ou de preuve liées à cette sémantique. En aucun cas, ils n'ont quelque compréhension des connaissances qu'ils enregistrent : toute méta-connaissance, ou compréhension des connaissances spécifiées, doit être programmée dans le système.

2.3. Techniques de représentation des connaissances.

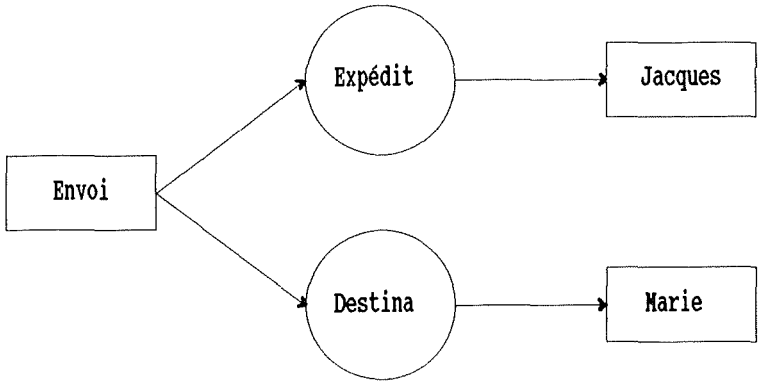
La représentation des connaissances par la logique, déjà évoquée au point 1 de ce chapitre, sera également présentée au points 3 et 4. Nous ne la détaillerons donc

pas ici. Mais nous voudrions présenter deux autres types de représentations des connaissances, alternatives à la logique, qui nous semblent aujourd'hui des axes importants d'application : les réseaux sémantiques et les objets.

Ces représentations ne sont cependant que des alternatives à la représentation logique. Tous les types de connaissances ne sont pas représentables par la logique habituelle et l'intelligence artificielle a alors besoin de logiques plus puissantes pour représenter ces connaissances. Nous présenterons ces logiques au chapitre 3.

2.3.1. La représentation réseau.

Cette représentation est principalement graphique. Un réseau sémantique est composé de plusieurs graphes conceptuels, chacun représentant une formule logique. Les graphes conceptuels comportent deux types de noeuds : un rond, qui est le prédicat, et des rectangles, qui sont les arguments. Ils sont reliés entre eux par des arêtes figurant la relation logique. A titre d'exemple, formalisons dans un graphe conceptuel la proposition ' jacques est l'expéditeur d'un envoi dont marie est la destinataire' :



(figure 1.2)

Cette proposition serait représentée en logique par la clause suivante :

expéditeur(jacques,envoi),destinataire(envoi,marie).

Un réseau sémantique, regroupant donc nombre de graphes conceptuels, décrit également les connexions entre les graphes conceptuels et les immerge dans leur contexte qui est le domaine du discours [Thayse 89]. Pour cela, le réseau part d'un graphe conceptuel simple puis ajoute d'autres graphes en les interpénétrant, c'est-à-dire en ne dupliquant jamais un élément mais en enrichissant le graphe en cours des relations du graphe supplémentaire; cette opération demandant que ces graphes aient des arguments en commun.

Le contexte est introduit après cela, grâce à un prédicat 'instantiation', signifiant pour tout objet[7] quel est son type. Les quantificateurs peuvent également être introduits dans les rectangles, spécifiant si l'objet est unique ou générique.

Souvent, un réseau sémantique aura une structure hiérarchique car la hiérarchie est souvent un mode de représentation aisé. De cette représentation découleront alors dans le réseau des propriétés héritées que l'on aura attachées à des noeuds génériques.

2.3.2. La représentation objet.

Dans une représentation objet, on rassemble les formules logiques contenant les mêmes instances en structures plus larges, appelées unités ou cadres (en anglais : frames) [Thayse 89] . Plutôt que de créer différentes formules indépendantes, on va réaliser une structure regroupant toute l'information relative à un même objet.

Une unité comprendra toutes les relations ayant trait à un même objet. Un cadre, quant à lui, traitera plutôt une relation en tant qu'objet, relation dont chaque attribut concernera un objet réel, appelé facette du cadre. Prenons deux exemples :

unités : jacques
expéditeur(jacques,envoi)
marie
destinataire(envoi,marie)

[7] C'est-à-dire pour tout rectangle, dans la représentation graphique.

cadre : envoi

expéditeur	jacques	(facette 1)
destinataire	marie	(facette 2)

Ces notations peuvent aussi être simplifiées, des éléments étant implicites. La quantification est introduite en remplaçant, p.ex., l'objet 'jacques' de la facette 1 du cadre par une variable. Mais pour effectuer un raisonnement sur les cadres, il faut choisir un mécanisme de base permettant d'accorder des cadres de données et le type de cadre formant la solution : c'est là la problématique des langages orientés objet, qui dépasse le cadre de ce travail. Nous retiendrons seulement qu'ayant été construite principalement par souci d'efficacité, cette approche manque de la rigueur de la logique en ce qui concerne les preuves et la déduction.

3. Prolog.

Historiquement, Prolog a été inventé pour être un langage capable de rendre directement exécutables des grammaires 'context-free'. Cette capacité découle de ce qu'une clause de Horn peut aussi être lue comme une règle de réécriture, donc comme une règle de production d'une telle grammaire. Ainsi, une grammaire exprimée logiquement pouvait être vue comme un programme, et le pas vers la programmation logique était fait dans l'esprit d'Alain Colmerauer. La transformation d'une grammaire 'context-free' vers Prolog est, à l'heure actuelle, tout-à-fait automatisée; nous avons utilisé cet acquis pour la définition du langage de commandes de AIDA.

Prolog a été développé à l'université de Marseilles par l'équipe d'Alain Colmerauer et sa première version a été disponible en 1973. Plusieurs dialectes en ont été tirés, mais ceux-ci ne se différencient guère que par des syntaxes différentes. Tout au long de cette présentation et dans notre application, nous avons utilisé la syntaxe d'Edinburgh, qui est la plus répandue. Pour une présentation en profondeur de Prolog, nous ne pouvons que renvoyer le lecteur à [Clocksin 87], ainsi qu'aux deux monuments destinés aux programmeurs en Prolog que sont [Bratko 86] et [Shapiro 87]. Le plan de la présentation qui va suivre s'inspire en outre du chapitre 2 de [Walker 87].

3.1.Caractéristiques de Prolog.

3.1.1. Programmation non procédurale.

Comme nous l'avons déjà dit, Prolog est un langage déclaratif. Les techniques habituelles de programmation ne sont donc pas applicables à Prolog bien que ses règles soient exécutées par un mécanisme d'inférences qui impose un certain ordre dans leur exécution. Contrairement à une exécution séquentielle, les règles de Prolog sont exécutées par 'backtracking'. C'est ce mécanisme de 'backtracking' qui induit chez les programmeurs une nouvelle façon de penser l'exécution de leurs programmes.

Prolog est étonnament simple. Il ne possède aucune structure de contrôle, aucun type et n'a même pas d'instruction d'affectation. En principe pourtant, Prolog est capable d'exécuter tout ce que l'on peut réaliser dans un autre langage de programmation car il peut, lui aussi, simuler une machine de Turing.

3.1.2. Faits et prédicats.

Un fait exprime une propriété d'un objet ou une relation entre plusieurs objets. Un fait s'exprime en appliquant un prédicat à une liste d'arguments. Sa forme est donc du genre voiture('peugeot 205'), précisant que la peugeot 205 est une voiture. Notons ici que le nom de la voiture a dû être mis entre apostrophes, le caractère 'espace' n'étant pas admis dans un nom d'objet ou de prédicat. Le nom de prédicat est quelconque, ainsi que le nombre de ses arguments. La définition d'une voiture pourrait aussi bien être voiture('peugeot 205',vert,5,ts42n4), précisant alors outre le nom de la voiture, sa couleur, le nombre de portes qu'elle possède et son numéro d'immatriculation. On remarque aussi que l'ordre des arguments est identifiant.

3.1.3. Variables et règles.

Tout mot commençant par une majuscule est considéré comme une variable, tandis que tout mot commençant par une minuscule est considéré comme une

constante. Une variable spéciale existe, notée '', et appelée variable anonyme : elle peut prendre n'importe quelle valeur et est toujours différente, même si elle apparaît plusieurs fois dans un même prédicat. Ainsi, là où pred(,) peut représenter pred(2,3), le prédicat pred(X,X) n'accepte que deux arguments identiques, bien que quelconques.

Des prédicats sont enregistrés tels quels dans un programme Prolog. Ils définissent ainsi des faits. Mais il est possible de définir des relations de manière plus complexe. En effet, si l'on veut faire une base de connaissances généalogiques, il serait très long d'entrer toutes les relations familiales sous leur forme explicite. On pourrait par exemple se contenter de définir chaque personne comme étant un homme ou une femme et comme étant le fils ou la fille de quelqu'un. Sur base de ces faits, toutes les autres relations parentales peuvent en effet être déduites.

Définissons les relations suivantes :

```
homme(marc).
homme(frédéric).
femme(véronique).
enfant(frédéric,marc).
enfant(frédéric,véronique).
```

A ces relations, on pourrait vouloir ajouter que frédéric est le fils de marc, et non sa fille ! Pour ce faire, Prolog permet d'écrire la règle suivante : fils(X,Y):-enfant(X,Y),homme(X).

Bien qu'elle paraisse évidente, la relation posant que pour être le fils de quelqu'un il faut être un homme et être l'enfant de cette personne est à la base de toute programmation en Prolog. Cette règle peut se lire ' X est le fils de Y si X est l'enfant de Y et si X est un homme'. Le lecteur voit que, grâce à de telles règles, il est très facile de définir les notions de père, de mère, de frère, de soeur, de cousin, d'époux, etc.

Dans l'écriture des règles, le signe ':-', que nous avons traduit par 'si', correspond en fait au connecteur logique <=. Le lien avec la logique devient alors explicite, surtout avec la virgule qui correspond au connecteur 'et'. Ces règles et faits

peuvent alors être considérés comme des clauses de Horn grâce à l'hypothèse du 'ou' implicite.

En effet, les deux règles 'homme' que nous avons vues pourraient s'écrire homme(X) :- homme(marc) ; homme(frédéric). Le point-virgule représente le connecteur 'ou'. La syntaxe Prolog permet en effet de réécrire cette règle en :

homme(X):-homme(marc).
homme(X):-homme(frédéric). .

Ainsi, le 'ou' est implicite entre des règles ayant la même partie gauche du signe ':-', donc ayant le même nom de prédicat. Cela vaut évidemment aussi pour les faits ayant même prédicat[8]. Les autres opérateurs sont aussi disponibles en prolog : '~' est notée 'not' et se place devant tout prédicat, l'équivalence A<=>B est traduite par la règle A:-B, B:-A.

3.1.4. Les buts:

Les faits et les règles appartiennent au programme Prolog mais ne sont pas exécutables. On ne peut lancer une action qu'en entrant un but. Un but simple est par exemple un fait : le but est vérifié si le fait est trouvé dans le programme. Ainsi, le but ?- homme(marc) se voit répondre 'yes' par Prolog tandis que le but ?- homme(jean) se voit répondre 'no', aucun fait ne disant que 'jean' est un homme. Une réponse négative signifie donc que le but ne peut être prouvé sur base des connaissances enregistrées, et ne signifie pas nécessairement que le but est faux. En conclure que 'jean' n'est pas un homme serait donc erroné.

Un but un peu plus compliqué serait ?- homme(X). Ici, Prolog doit chercher toutes les constantes unifiables avec X pour qu'elles vérifient la relation 'homme'. Dans notre cas, Prolog répond 'marc' car c'est la première constante rencontrée qui prouve le but demandé, puis, si l'utilisateur tape un ';', un 'ou', pour demander à Prolog une

[8] Le prédicat, ou nom de la relation, est parfois aussi appelé foncteur.

nouvelle solution, l'interpréteur répond 'frédéric'. Un troisième ';' amène la réponse 'no', aucun fait appartenant au programme ne pouvant plus satisfaire le but.

Prenons comme autre exemple la question suivante : ?- enfant(X,Y). Pour prouver cela, Prolog va parcourir sa liste de règles et de faits de haut en bas et chercher le premier prédicat 'enfant'. Il trouve donc X=frédéric et Y=marc. Poursuivant sa requête après un ';' entré au clavier, il trouve X=frédéric et Y=véronique. Un nouveau ';' mettra Prolog dans l'obligation de répondre 'no'.

3.1.5. Les structures de Prolog.

La forme générale des clauses[9] Prolog que nous avons vues est donc toujours 'tête := corps .', avec le cas particulier du fait qui est de la forme 'tête .' . La tête comporte donc toujours un prédicat et au moins un argument. Le corps peut posséder un nombre quelconque de prédicats et d'arguments, les prédicats pouvant être séparés par l'opérateur 'et' ou par l'opérateur 'ou'. Un prédicat du corps peut également être nié par application de l'opérateur 'not', mais son utilisation est interdite en tête d'un prédicat.

Tout ceci montre que Prolog est un langage relationnel et non fonctionnel comme Lisp, par exemple. En effet, Prolog ne calcule jamais aucun résultat, sauf lorsque certains opérateurs de calcul sont utilisés. La notation fonctionelle utilisée en Prolog ne sert donc qu'à représenter des structures de données et non à calculer une quelconque fonction. Cette notation entraîne aussi une structure quasi unique de données : l'arbre.

3.1.6. Les prédicats standards de Prolog.

Etant un langage purement déclaratif, Prolog n'a théoriquement pas besoin de prédicats standards. Cependant, certaines applications les nécessitent : l'amélioration des performances lors des manipulations arithmétiques ou de caractères, une facilité

[9] Une clause Prolog est soit une règle soit un fait.

accrue dans l'emploi de certaines opérations et, surtout, l'accès aux entrées/sorties et au système d'exploitation.

Prenons un exemple de 'pur Prolog' : le calcul récursif d'une factorielle. Les clauses de calcul d'une factorielle sont les suivantes :

```
factorielle(0,1).
factorielle(N,X) := M is N-1, factorielle(M,Y), X is Y*N.
```

L'opérateur important ici est 'is'. Il permet de forcer le calcul d'une expression arithmétique et d'associer le résultat à une variable. Contrairement au cas d'unification standard de variable avec une constante, l'association arithmétique ne peut pas être défaite pour chercher une nouvelle solution. Ainsi, ce petit opérateur de deux lettres introduit un concept très imporatnt de Prolog : le déterminisme. ce concept interdit le backtracking. Sur sa base, les clauses Prolog peuvent être classées en clauses déterministes (et donc en fait procédurales) et en clauses non déterministes, respectant alors l'aspect purement déclaratif de Prolog.

L'opérateur 'is' fait partie des prédicats appelés non-logiques, ou extra-logiques. Ces prédicats appartiennent à trois catégories : la première catégorie comporte les prédicats qui contrôlent l'exécution de Prolog, comme le 'is' qui interdit un 'backtracking' ; la deuxième catégorie contient les prédicats gérant dynamiquement l'espace de travail de Prolog et permettant notamment d'ajouter ou de retirer dynamiquement du programme en cours d'exécution des règles et des faits; la troisième catégorie rassemble les prédicats système qui offrent l'accès aux entrées/sorties et au système d'exploitation.

Ces prédicats extra-logiques ne sont utiles que par leurs effets de bord, c'est-à-dire par les données ou par les structures de données qu'ils changent ou auxquelles ils accèdent en dehors de la logique de Prolog. Un exemple typique d'un tel prédicat est la sortie d'un résultat sur papier ou la gestion d'un curseur d'écran.

3.1.7. Le moteur d'inférences

Si, déclarativement, les clauses Prolog sont vraies ou fausses, procéduralement, les buts réussissent ou échouent. L'effet procédural provient de la combinaison de la logique et du moteur d'inférences. Pour voir son fonctionnement, supposons les règles suivantes :

père(X,Y) :- enfant(Y,X) , mâle(X).
enfant(frédéric,marc).
mâle(frédéric).
mâle(marc).

Supposons que Prolog doive prouver le but ?- père(A,B). Une telle question sortira normalement tous les pères définis dans la base de connaissances. Voyons comment Prolog s'y prend.

La question est ?- père(A,B). Prolog recherche la première clause ayant même prédicat : c'est père(X,Y):-enfant(Y,X),mâle(X). . Il unifie alors la variable X à A, et Y à B, ce qui ne change théoriquement rien mais crée un lien dynamique qui pourra être remplacé plus tard, si nécessaire, par un autre lien. La question est devenue père(X,Y) sachant que pour cela, il faut vérifier enfant(Y,X) et mâle(X).

Prolog parcourt à nouveau le programme pour enfant(Y,X) et trouve enfant(frédéric,marc). A nouveau, il unifie Y à la constante frédéric et X à la constante marc. La clause 'enfant' ayant réussi, il faut prouver mâle(X), qui est devenu mâle(marc) depuis. Ce fait est trouvé deux lignes plus loin dans le programme. Notons que le fait mâle(frédéric) n'est pas utilisé !

Les deux clauses nécessaires pour 'père' ayant été prouvées, Prolog répond à la question père(A,B), sachant A==[10]X==marc et B==Y==frédéric, par ' yes : A = marc and B = frédéric'. Si un autre prédicat père existait dans le programme, Prolog aurait défait l'unification de A à X et de B à Y, lors du 'backtracking' pour la recherche d'une seconde solution et aurait pu les unifier aux autres variables définissant l'autre prédicat père. Cette première unification, bien qu'elle semble inutile, est donc très

[10] Nous utilisons le double == pour représenter la relation d'unification. Cette notation n'est pas standard.

importante. Elle permet ainsi l'évaluation des buts 'ou', puisqu'une deuxième définition du prédicat père est en fait un 'ou' pour celui-ci.

L'unification de deux variables entre elles peut être vue comme la définition, par le moteur d'inférences, d'un point de 'backtracking', c'est-à-dire d'un point à partir duquel une nouvelle branche naît dans l'arbre des preuves possibles. Selon cette vue, une solution est alors l'ensemble de toutes les feuilles vraies d'une branche de l'arbre. Le lecteur aura remarqué la nécessité d'un mécanisme aussi puissant que l'unification étant donné qu'une instantiation d'une variable à une autre n'aurait aucun sens.

3.2. L'aspect déclaratif de Prolog.

C'est le coeur du langage Prolog. Il contient toute la force d'une implémentation de la logique prouvable sans aucune concession à une quelconque vue procédurale. Cet aspect offre des notions comme l'invertibilité, les spécifications exécutables et la manipulation symbolique. Il offre aussi la structuration de prédicats, comme nous l'avons vu avec la définition d'une adresse, et la structure de données de base qu'est la liste.

Ajoutons un petit mot ici sur les listes, dont nous n'avons pas encore parlé. Une liste est une collection ordonnée d'éléments quelconques mais d'arité[11] identique. Une description d'une liste se fait toujours grâce à l'opérateur '|', selon la forme : ['tête de liste' | 'liste queue']. La tête de la liste est toujours un prédicat unique tandis que la queue de la liste est de longueur quelconque, voire nulle. Le lecteur trouvera, sous la rubrique 'utilitaires divers' de notre programme, des clauses de gestion de listes. Il remarquera notamment que la liste est la seule structure offerte par Prolog permettant de regrouper plusieurs données; elle est extensivement utilisée par notre application.

3.3. L'aspect procédural de Prolog.

Cet aspect du langage oblige le programmeur à tenir compte de l'ordre d'exécution des clauses Prolog. Il s'agit bien maintenant d'exécution puisque certains

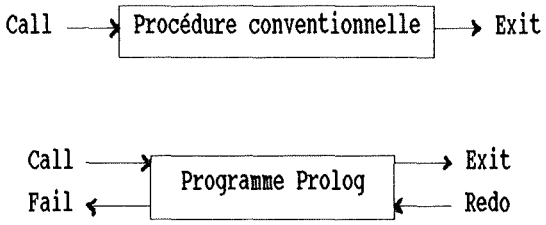
[11] L'arité d'un prédicat ou d'un objet est le nombre de ses arguments.

prédicats utilisés ici sont déterministes, de sorte que leur effet ne peut pas être 'annulé' lors d'un 'backtracking'.

3.3.1. Le backtracking et le cut.

Le procédé de backtracking fait partie du moteur d'inférence. Nous avons déjà essayé de le présenter plus ou moins intuitivement; voyons maintenant concrètement en quoi il consiste.

Le backtracking est un enrichissement du procédé habituel d'appel de procédure. Il permet à des règles, que l'on a appelées et dont on est sorti comme dans des procédures normales, d'être automatiquement réactivées quelque temps plus tard pour résoudre à nouveau un sous-problème qui n'a pas conduit à une solution satisfaisante. Le schéma suivant montre les vues procédurales et en Prolog de cette dynamique d'appel :



(figure 1.3)

Dans le cas de Prolog, si une règle peut se terminer par un exit, et donc par un succès, elle peut également se terminer par un 'fail', par un échec. Le schéma nous montre bien que le flux gauche-droite n'est plus respecté en cas d'échec, et qu'une autre philosophie d'appel est choisie : le programme continue, mais en réactivant le but précédent par un 'redo'. Prenons un exemple simple :

```
p :- q , r , s , t.
p :- u , v.
```

Une fois le but ?- p invoqué, Prolog cherche à prouver q, r, s et t. Supposons que q et r réussissent, mais que s échoue. Le contrôle est alors rendu à r que l'on relance pour lui demander une nouvelle solution (r aura donc été un point de 'backtracking' possible). Si, à nouveau, s échoue et que r ne peut générer aucune nouvelle solution, le contrôle est donné à q. Si, au total, la première règle ne parvenait pas à prouver p, la deuxième serait alors appelée selon le même principe. Mais, à l'inverse du 'backtracking' pour les ',', si la deuxième règle échoue, parce qu'étant un 'ou', le 'backtracking' ne peut pas remonter à la première règle, et le prédicat p échoue complètement.

C'est donc la conjonction des 'fail' et des 'redo' qui forme le mécanisme de 'backtracking'. Mais ce mécanisme contient un mécanisme de contrôle, appelé 'cut' et noté '!', qui permet de supprimer des options 'ou'. La signification du 'cut' en lui-même est très simple : dès que le moteur d'inférence en rencontre un, toutes les solutions du prédicat en cours trouvées jusqu'à présent sont gelées et il est impossible de générer, par 'backtracking', une nouvelle recherche sur ces prédicats.

Soit le prédicat p :- q, r, !, s. Si r échoue, le 'backtracking' peut relancer q, mais si c'est s qui échoue, aucune relance n'est plus possible et c'est p tout entier qui échoue. Le 'cut' est donc très puissant; il est utilisé en général pour modéliser dans un prédicat l'idée suivante : si on est arrivé jusqu'ici dans la recherche, alors on a trouvé la seule solution possible et si, par la suite, elle s'avère ne pas être la bonne, on est sûr de ne pas en trouver d'autre. Inutile de dire qu'un usage non réfléchi de ce prédicat de contrôle peut amener Prolog à ignorer beaucoup de solutions. De plus, le 'cut' rend automatiquement déterministe le prédicat dans lequel on l'utilise !

3.3.2. Les autres aspects procéduraux.

Tous les autres aspects procéduraux de Prolog sont basés sur des prédicats ayant des effets de bord. Tous sont nécessités par un style de programmation procédural. En voici quelques exemples.

3.3.2.1. Les résultats intermédiaires.

Supposons des clauses de calcul d'une série de Fibonacci. Le calcul de cette série nécessite la rétention de deux résultats intermédiaires lors de chaque calcul effectué. Pour retenir ces deux résultats, alors qu'ils ont déjà été calculés, on les ajoute dynamiquement au programme jusqu'à ce qu'ils soient utilisés dans un calcul ultérieur. Cette façon de faire est nécessitée par la longueur du calcul, qui cause très vite un débordement de pile si on demande au prédicat récursif de recalculer à chaque fois tous les résultats intermédiaires.

Cette méthode apporte aussi un gain non négligeable dans le temps d'exécution du prédicat. De plus, pour ne retenir qu'un seul résultat intermédiaire, on fait plus souvent appel au mécanisme de l'accumulateur, que nous avons déjà présenté.

3.3.2.2. Les prédicats d'entrée/sortie.

Ces prédicats sont utilisés dans la construction de l'interface du programme Prolog avec l'extérieur ou avec d'autres langages. L'interface avec l'extérieur comporte la gestion de l'écran, du clavier, de l'imprimante et des fichiers de clauses. Dans ces derniers, on ne peut guère véritablement parler de fichier. Toutes les clauses du programme sont en mémoire et il est seulement possible d'en sauvegarder quelques-unes ou d'en charger des supplémentaires.

Souvent, l'environnement Prolog accepte l'interfaçage avec d'autres langages. Bien entendu, les procédures qui devront être considérées comme des clauses ne pourront que réussir, le programmeur devant gérer lui-même les codes d'erreurs et les traduire en Prolog par les échecs et les réussites correspondants. Cependant, dans la version développée par Siemens, il est possible d'écrire des prédicats pouvant échouer en C puis de les recompiler avec tout l'interpréteur Prolog. En fait, cela revient à ajouter un nouveau prédicat à Prolog. Ce mécanisme, très lourd, n'est pas du tout à la portée d'un débutant.

4. Prolog pour les bases de données et les bases de connaissances.

4.1. Introduction.

Dans cette partie, nous désirons présenter la position de Prolog face aux bases de données et aux bases de connaissances. A.I.D.A., le prototype que nous avons construit et qui sera présenté au chapitre 2, est un système à base de connaissances destiné à l'exploitation de 'dumps' physiques sur BS2000. Nous utiliserons des connaissances tirées de ce prototype comme exemples de cette discussion. Mais avant de discourir sur Prolog, rappelons la différence existant entre une base de données et une base de connaissances.

Les bases de connaissances sont des systèmes possédant une théorie sous-jacente leur permettant d'interpréter l'information qu'elles contiennent d'une certaine manière[12]. Les bases de données, elles, n'ont pour but que d'enregistrer et de fournir des quantités parfois importantes de données; ces opérations doivent être exécutées avec célérité et un maximum de sécurité pour les données. La différence de finalité est dès lors évidente : les bases de connaissances ont une théorie de la sémantique de leurs données et ne se préoccupent pas des techniques d'enregistrement de ces dernières, tandis que les bases de données sont concernées par tous les problèmes d'accès à des données, quelle que soit la sémantique de ces dernières.

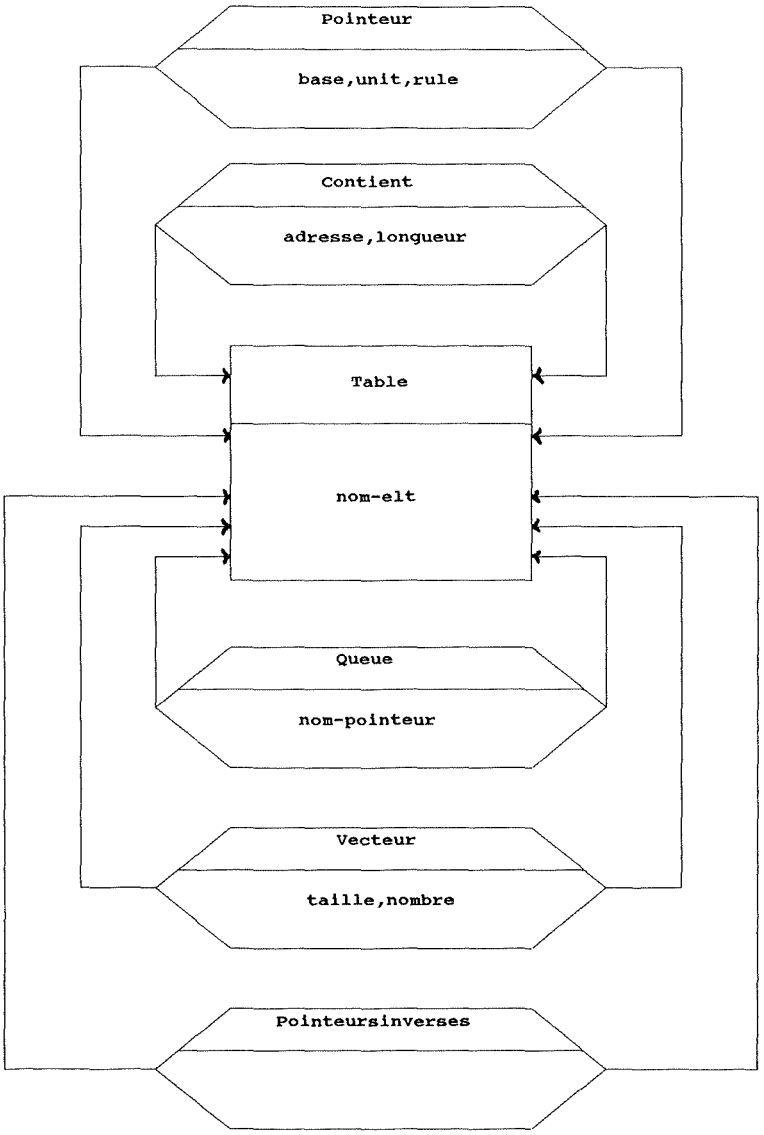
Le lecteur peut déjà deviner que Prolog est concerné par les deux aspects que nous venons d'évoquer. Le modèle des données de A.I.D.A. que nous allons présenter est un modèle relationnel sur lequel le prototype vérifie sans cesse, en recherchant des chemins, la cohérence des relations décrites. Ces dernières sont donc traitées à la fois comme des données et comme des connaissances.

4.2. Schéma des données.

Nous présentons ci-après le schéma entité/relation du modèle de données de notre prototype. Le lecteur aura remarqué que pour Prolog, une base de connaissances n'est qu'une liste de faits considérés comme atomiquement vrais.

---

[12] Bien que dans le cas de Prolog, cette manière ne soit pas toujours certaine du point de vue du programmeur.



(figure 1.4)

Les déclarations qui suivent montrent comment ces connaissances peuvent être emmagasinées dans un schéma relationnel.

table (nom-elt)[13]
contient (nom-elt , nom-elt2 , adresse , longueur)
queue (nom-elt , nom-elt2 , nom-pointeur)
pointeur (nom-elt , nom-elt2 , base , unit , rule)

[13] Un attribut souligné signifie que ce dernier doit être identifiant.

vecteur (nom-elt , nom-elt2 , taille , nombre)
pointeursinverses (nom-elt , nom-elt2)

4.3. Prolog et les bases de connaissances.

Par définition, Prolog est adapté aux bases de connaissances puisque son but est d'ajouter de la sémantique à des données. Toute la signification du mécanisme des prédicats et de la preuve tend vers ce but.

Nous pouvons dès à présent citer quelques avantages et inconvénients inhérents à ce langage.

4.3.1. Inconvénients.

Prolog n'implémente pas purement la logique des prédicats du premier ordre. Il manque notamment la définition d'une vraie négation, qui serait prouvable, et des clauses de Horn purement récursives. Ces limites entraînent beaucoup de compromissions de la part de l'application construite. Nous pouvons citer ici en guise d'exemple la nécessité d'utiliser des prédicats extra-logiques tels 'findall' afin de s'assurer la complétude de certaines requêtes, ou encore la nécessité de jongler avec deux négations différentes; l'une de celles-ci est standard et offerte par Prolog et l'autre n'est généralisée que par une gestion interne propre à l'application.

La base de données interne de Prolog est très limitée. Nous avons déjà signalé comme le simple retrait séquentiel des prédicats demandés, l'absence des fonctions offertes par un gestionnaire de bases de données et enfin la particularité très contrariante d'ignorer si deux demandes d'accès identiques donneront le même résultat dans des contextes différents.

4.3.2. Avantages.

Regroupés sous la notion de 'closed-world assumption', les inconvénients de la négation et du séquencement des prédicats peuvent également être vus comme des

avantages car ils permettent à une application de raisonner sur des connaissances incomplètes, ce que le lecteur comprendra très vite en parcourant l'insignifiante taille de nos bases de tests pour notre prototype.

Les prédicats d'assertion et de rétraction dynamique de connaissances, bien que primaires et peu efficaces, nous ont cependant aidé grâce à la nature de leur comportement, fort proche de la manière humaine de raisonner. La recherche de tous les chemins, leur mémorisation et la détection parmi ces chemins de ceux vérifiés au niveau des occurrences en est un bon exemple.

Bien entendu, ce langage est très puissant pour faire des preuves et des tests; nous nous sommes largement basés sur ses possibilités pour développer notre prototype.

Nous noterons aussi la possibilité, facilement implémentable et d'ailleurs déjà partiellement implémentée, d'écrire des prédicats de type 'how' et 'why', c'est-à-dire expliquant comment ou pourquoi le système d'inférence en est arrivé à certaines conclusions sur base des connaissances.

Nous ajouterons simplement à ce sujet que notre prototype, grâce au choix de son langage d'implémentation, nous a offert presque gratuitement à la fois une grande indépendance par rapport aux données spécifiant l'architecture de structures étudiées et un langage d'interrogation particulier de très haut niveau.

4.4. Prolog et les bases de données.

Nous commencerons cette présentation en soulevant une tendance actuelle révélée par plusieurs de nos lectures, dont [Li 84] : l'utilisation de plus en plus courante de la logique comme fondement théorique des développements récents faits en matière d'approfondissement et d'extension du modèle relationnel et parfois plus généralement en matière de théorie des langages d'interrogation de bases de données. Nous pensons utile d'y déceler une acception implicite de la richesse que ce modèle peut apporter à l'exploitation des bases de données. Cette vision est très compréhensible si nous tenons compte de l'architecture conceptuelle offerte par la logique pour la modélisation de concepts de bases de données ainsi que de l'outil direct

pour les implémenter qu'elle offre via Prolog. Ce formalisme, naturel dans ce cadre, est encore plus riche de par ses capacités naturelles d'écriture de compilateurs ou d'interpréteurs. Mais essayons d'argumenter quelque peu au sujet de cette affirmation.

Nous avons remarqué, notamment chez [Lucas 88], que le langage d'interrogation SQL, basé sur le modèle relationnel, était implémentable assez directement en Prolog. Cet auteur démontre, par une recherche relationnelle de faits couplée à Prolog, que les traits caractéristiques d'un bon langage d'interrogation relationnel sont parfaitement implémentables; il montre en outre que le temps d'exécution supplémentaire requis par Prolog pour les accès directs aux données est de complexité linéaire et non exponentielle comme on pourrait s'y attendre au vu des techniques d'implémentation du langage Prolog lui-même.

Les traits importants implémentés avec succès par l'auteur sont le langage d'interrogation, la manipulation des données, l'écriture d'applications, la définition formelle de modèles de données et l'ajout de sémantique à ces dernières. Comme preuve de faisabilité de cette approche dans un système de taille réelle, ce qui n'est pas encore le cas de notre prototype, nous retiendrons l'application du système de [Lucas] à la peinture en chaîne de voitures neuves, en temps réel, sur un ordinateur non optimisé pour l'exécution de Prolog.

L'approche générale d'utilisation de Prolog dans le domaine du relationnel consiste à utiliser Prolog comme un résolveur de problèmes. Coeur des applications et de toute la sémantique, ce résolveur fera appel à un système relationnel pour les simples accès aux données dont il a besoin. Appliquée sur une grande échelle grâce à la complexité ajoutée linéaire et avec des temps de réponse du système relationnel acceptables, cette idée pourra changer le système lui-même en un langage de prototypage et de développement, complet et surtout très rapide, d'applications orientées relationnel.

Mais une autre approche de l'utilisation de Prolog est envisageable, dans le sens d'une extension de ce dernier.

En effet, à condition d'y faire quelques ajouts, Prolog peut être un bon outil de construction d'une base de données relationnelle. Les axes actuels de recherche que l'on peut trouver dans la littérature sont les suivants :

a) Utilisation de Prolog comme optimiseur de requête. Après avoir décodé la requête et l'avoir transformée en formalisme logique, Prolog peut ainsi mettre en tête de requête ce qui génère le moins de non-déterminisme ou ce qui a le plus de chance d'échouer; il peut aussi sauter les clauses indépendantes de celles qui ont échoué lors du 'backtracking', et, bien entendu, indexer tous les accès.

b) Transformation de Prolog en un langage purement déclaratif, sans plus aucune sémantique procédurale ni de suppositions du genre 'closed-world assumption', et implémentation dans ce langage d'un compilateur de requêtes selon le sens précédemment décrit.

c) Extension des prédicats 'assert' et 'retract' afin d'en uniformiser la sémantique dans les différentes implémentations du langage et d'en étendre la sémantique pour pouvoir les 'défaire' lors d'un 'backtracking'. De plus, leur extension permettra une vérification de contraintes d'intégrité en autorisant un retour en arrière jusqu'à un certain point de vérification et supportera la concurrence et la problématique de la reprise d'incidents [Naish].

Dans cette dernière proposition, nous pouvons voir la volonté des auteurs de supporter dans Prolog même la notion, chère aux bases de données, de transaction.

Certains auteurs vont même plus loin et proposent, notamment dans [Maier], l'extension de Prolog en un langage de requêtes de second ordre, c'est-à-dire pouvant comprendre des requêtes rédigées dans un langage quasi naturel, traduisant cette requête en logique, l'optimisant et la passant à un système relationnel (éventuellement aussi écrit en Prolog).

Sans aller aussi loin dans les exigences requises de Prolog, il existe des techniques de programmation particulières à Prolog qui permettent de l'utiliser efficacement pour une application de base de données. Ces techniques ne sont utiles que dans une application assez restreinte, mais elles ont le mérite de fonctionner correctement. A ce sujet, nous renvoyons le lecteur à la notion de 'lemme' développée dans [Hepburn 87].

Bien que cette dernière affirmation relève plus du voeu pieux que d'une réalité concrète, nous estimons fondée notre raison de croire en Prolog dans ce domaine, tout en restant conscient de ses principaux avantages et inconvénients.

4.4.1. Avantages

+ le relationnel est directement implémentable en Prolog,
+ Prolog est un langage hautement extensible tant en fonctionnalités intrinsèques qu'en possibilités syntaxiques (notamment grâce à l'emploi des opérateurs),
+ les vues relationnelles forment un des mécanismes de base des prédicats Prolog,
+ au niveau de ses expressions, Prolog est plus puissant que l'algèbre relationnelle [Maier],
+ les contraintes d'intégrité sont directement implémentables en Prolog,
+ Prolog est un langage très aisé à utiliser pour échanger ou translater des données.

4.4.2. Inconvénients

- Prolog manque de possibilités de gestion de mémoires secondaires,
- Prolog ne supporte pas encore les notions de concurrence, de reprise, d'autorisation et de transaction en bases de données,
- Prolog ne possède pas de types de données,
- à l'exception du coûteux prédicat 'setof', Prolog ne supporte pas la notion d'ensemble,
- Prolog pose le problème de la génération de solutions non uniques,
- enfin, la syntaxe de Prolog, ainsi que sa philosophie, ne sont pas du tout évidentes pour un non initié.

5. Notre application.

Nous allons présenter ici les caractéristiques de Prolog qui nous ont amené à le choisir comme langage d'implémentation pour notre application. Bien sûr, ce choix a également pris en considération les particularités de notre application. Voyons quelles étaient ces particularités.

La tâche à effectuer se divisait en deux : réaliser une base de définitions des structures internes au système d'exploitation puis implémenter une application permettant de voyager à travers ces structures dans un 'dump'. La première tâche nous apparut la plus importante, l'application étant basée sur le choix des définitions.

Ces connaissances sur les structures étant les plus importantes, nous avons choisi un langage où toute connaissance était explicite. L'idéal, qui nous fut donné par Prolog, était de posséder des spécifications directement exécutables. Et, par extension, ces spécifications exécutables nous offraient également les capacités d'explication du raisonnement suivi qui nous étaient nécessaires.

Ainsi, le choix était posé. Mais tous les aspects positifs de notre langage déclaratif ne supplantaient pas un inconvénient majeur : la nécessité d'utiliser un langage procédural pour accéder aux 'dumps'. Heureusement, comme l'interpréteur de Prolog était écrit en un langage procédural, des passerelles existaient.

Les avantages offerts par Prolog ont eu de nombreux impacts sur notre travail. Le plus grand avantage était lié au fait que programmer en Prolog consiste à décrire les connaissances possédées, et non à décrire une solution particulière. Cela amène en général [Lazarev 88] cinq à dix fois moins de lignes de code en Prolog, donc moins de temps de développement, plus de facilités, des vérifications plus aisées, de grandes possibilités de prototypage rapide et une maintenance plus aisée.

La modularité naturelle de Prolog nous a permis, tout au long du travail, de développer plusieurs parties de l'application en même temps, ce qui n'est possible dans un langage procédural qu'avec une plateforme de génie logiciel appropriée. Le concept

d'invertibilité[14] a aussi été crucial dans notre prototype; le lecteur pourra le voir au point 3.2.1b. du chapitre 2.

Mais Prolog n'a pas que des points forts. L'inconvénient majeur de ce langage, qui nous a d'ailleurs vite interpellé, est sa lenteur. L'interpréteur n'y était bien sûr pas pour rien. Nous avons heureusement pu utiliser des techniques comme le cut, les accumulateurs[15] ou les listes différentielles [Shapiro 86] pour optimiser le temps d'exécution des prédicats le plus souvent utilisés.

---

[14] Ce concept de Prolog offre l'utilisation de tout paramètre de Prolog en entrée ou en sortie, au choix du programmeur. La programmation doit s'assurer du non-déterminisme du prédicat, mais le concept offre l'utilisation d'un intégrateur de fonctions, p.ex., pour les dériver.

[15] Un accumulateur est un paramètre supplémentaire permettant de stocker des données temporaires lors du travail d'un prédicat, offrant alors la possibilité de tester en cours d'exécution la solution construite sans avoir à attendre que le mécanisme d'inférences se trouve dans un chemin sans issue et backtracke.

Chapitre 2 : A.I.D.A. : an Artificially Intelligent Dump Analyser.

1. Les besoins

La firme Siemens, qui m'a accepté comme stagiaire pendant quelques mois, développe notamment un système d'exploitation appelé BS2000, dont la taille avoisine les 10 méga-octets. Dans le cadre du développement et de la maintenance de ce logiciel de taille géante, elle rencontre de nombreux problèmes, et beaucoup de ceux-ci sont dûs à cette taille. C'est en espérant remédier à un certain nombre de ces difficultés qu'elle m'a demandé de résoudre le problème suivant.

Pendant le développement d'un sous-système pour le système d'exploitation, la phase de test consiste principalement en l'analyse des structures de données sur plusieurs niveaux. Cela veut dire que le développeur doit pouvoir voyager à travers des pointeurs, des queues, des vecteurs et d'autres structures afin de pouvoir atteindre une information dans une structure de données bien précise.

Des produits donnant cette possibilité sont déjà disponibles chez Siemens; nous avons, par exemple Helga, Aidsys et Damp qui connaissent une petite partie des structures de données du système. Ces produits sont notamment capables d'atteindre un certain processus lancé par le système d'exploitation. Mais cette connaissance est limitée et, n'étant pas programmée comme connaissance explicite dans ces programmes, n'est disponible à aucun développeur de sous-système. Toute modification aux structures de données du système d'exploitation ou de ses sous-systèmes implique aussi une correction dans le produit lui-même. De plus, le produit adapté à une nouvelle version du système d'exploitation n'est disponible qu'un certain temps après cette nouvelle version, car l'ajout de connaissances supplémentaires dans ces produits signifie la construction d'autres modules ou de modules différents et nécessite dès lors d'autres délais d'adaptation dûs au cycle de vie de tout logiciel.

D'un autre côté, si les développeurs peuvent voyager parmi leurs propres structures de données lorsqu'elles sont connues par les produits qu'ils utilisent, bien qu'un tel parcours ne puisse être effectué par le développeur qu'avec un effort devant être considéré comme excessif, ces produits ne leur permettent pas d'avoir accès à toutes les structures de données appartenant au système. Dans certains cas, des projets sont retardés simplement à cause d'un manque de possibilités d'accès à un élément dans une certaine structure de données.

Ce problème pourra être résolu par un programme permettant de voyager dans n'importe quelle structure de données en utilisant une description formelle de la structure de ces dernières. Grâce à un langage concis de description de structures de données, le développeur d'un nouveau sous-système sera à même de communiquer à l'avance la description des structures qu'il implémentera et ainsi de maintenir constamment à jour la base de connaissances du programme 'parcoureur de structures'.

Ce 'parcoureur' pourra également être utilisé pour modifier des informations enregistrées dans les structures de données elles-mêmes, tout en vérifiant la validité d'une telle opération, et deviendrait ainsi un outil de prototypage rapide. La simplicité d'une telle approche apparaît alors garante de son efficacité.

L'outil développé devra également être à même d'effectuer son travail sur une représentation figée du système, telle que celle fournie par un 'dump', et pourra également être utile à tout utilisateur pour rechercher les incohérences du système. Il va de soi que le travail d'interrogation des structures de données par un utilisateur devra être réduit au minimum, n'exigeant de ce dernier qu'une connaissance de base du système d'exploitation dans son ensemble.

Son intérêt ne se portant que sur une partie bien particulière du système d'exploitation, un développeur refusera, légitimement, de s'ouvrir aux détails des structures se trouvant entre les structures générales de base du système qu'il utilise et celles qu'il développe. Ainsi, le programme devra lui permettre d'accéder à ses propres structures de données, mais la manière d'atteindre ces structures devra pouvoir lui être indifférente. Dans ce même cadre d'idées, le programme devra être susceptible d'expliquer pourquoi une certaine structure n'aura pas pu être atteinte.

Si le langage de spécification s'avère assez puissant, l'outil de parcours pourra, dans une étape ultérieure, être intégré dans le système d'exploitation même et ainsi fournir ses services de recherche à quiconque en éprouvera le besoin. Cela apportera également une amélioration dans la modularisation du système d'exploitation.

Le prototype que nous avons réalisé implémente ces idées pour une partie du système d'exploitation, appelée NUCLEUS ou BASYS, et ce, dans un nombre limité de structures de données de ce noyau. Le lecteur trouvera une description du noyau dans [Dedié 88]. Notre but étant de construire un programme qui rencontre les besoins exprimés, nous vous présentons ci-après nos développements.

2. Les spécifications

Les spécifications couvriront les deux aspects demandés au projet : d'une part la définition d'une représentation formelle pour les connaissances portant sur les structures de données et leur organisation, d'autre part la construction d'un programme utilisant ces connaissances afin d'offrir à quiconque les fonctionnalités présentées dans les besoins.

2.1. La modélisation des structures de données

2.1.1. Les structures de données simples

La structure de base de tout le système d'exploitation est la table. Il s'agit d'une structure contenant des données définissant l'état du système. Toute table est composée d'éléments de différents types. Chaque élément d'une table est accessible par son nom, ce dernier devant toujours être identifiant. Dans le système d'exploitation, certaines tables peuvent exister en plusieurs exemplaires, amenant le programme à devoir reconnaître et gérer la notion d'occurrence d'une structure de données.

Un élément d'une table peut très bien être lui-même une structure de données. Mais les composants de cet élément ne sont pas éléments de la table. Ainsi, par exemple, une table contenant un vecteur possède, par définition, l'élément formé du vecteur tout entier mais les éléments formant ce vecteur ne sont pas considérés comme des éléments de la table, et ne sont donc pas directement accessibles de la table à partir de leur nom.

2.1.2. Les structures de données complexes

Ces structures sont en fait des multiples d'autres structures. Trois types ont été retenus.

Le vecteur : c'est un ensemble d'éléments ayant tous la même structure. Le nombre de ces éléments est toujours connu, ainsi que leur taille. Tout élément peut donc séparément être indexé[1] à l'intérieur du vecteur lui-même.

La queue : c'est également un ensemble d'éléments mais dont chacun contient un pointeur vers l'élément suivant. La définition de ces éléments doit donc contenir implicitement la définition du pointeur. Seul, le premier élément d'une queue est considéré présent dans la table contenant la queue. Si d'autres doivent l'être, ils pourront être atteints par l'intermédiaire de leur identifiant et en parcourant la queue via la chaîne de pointeurs.

La double queue : c'est une queue possédant en plus un pointeur dans chaque élément, permettant d'atteindre aussi l'élément précédent. Une telle queue peut donc être parcourue dans les deux sens.

2.1.3. Les relations entre structures de données

Les structures de données ne devraient compter qu'une seule relation entre elles : la relation 'contient', d'ailleurs valable pour toutes les tables. Cette relation permet de définir toute structure de données à partir du concept de table. Les structures pouvant être imbriquées, le lecteur verra rapidement qu'une telle construction aboutit à une structure d'arbre. Cependant, dans la structure actuelle de nos spécifications, les éléments de données simples doivent toujours être des tables. Les connaissances décrites devant être adaptées à un système d'exploitation, il nous a paru intéressant de particulariser certaines tables ne possédant qu'un seul élément qui de plus ne fait référence qu'à un seul autre élément : les pointeurs. Bien qu'il s'agisse d'une relation

[1] Par indexage, nous entendons la possibilité d'identifier univoquement tout élément d'un vecteur sur base d'une information supplémentaire à son nom : sa position dans le vecteur, qui est toujours relative au nombre d'éléments que contient ce dernier.

entre structures de données, le pointeur sera considéré comme une structure de données simple, à rajouter à celle de la table. Dans leur définition, les pointeurs devront avoir leurs caractéristiques de taille, de base et d'unité.

2.1.4. La notion de règle

La dynamique d'un système d'exploitation confère aux pointeurs certaines propriétés dynamiques qu'il convient de modéliser le mieux possible. Pour ce faire, nous introduisons la notion de règle pour un pointeur. Elle couvre plusieurs cas :

succès/échec : cette règle indique si le pointeur pointe toujours, s'il ne pointe jamais ou dans quelles conditions il pointe. Une condition sera toujours une expression de type mathématique.

pointeurs conditionnels : comme les structures pointées peuvent être variables, de nombreux cas de structures dynamiques de pointeurs se présentent dans un système d'exploitation. En permettant, via la définition de pointeurs conditionnels, de définir plusieurs fois le même pointeur, tous les cas de figures peuvent être supportés par nos connaissances. Il va de soi que plusieurs définitions d'un même pointeur ne peuvent jamais être contradictoires[2].

2.2. Le programme de parcours de 'dumps'

Le programme doit pouvoir parcourir la spécification formelle, en accord avec le 'dump' tant que ce dernier ne contient pas d'incohérences, et doit pouvoir présenter le contenu du 'dump'. Derrière ce but se cachent des fonctionnalités très diverses.

Pour spécifier le chemin, l'utilisateur doit pouvoir soit être aussi général qu'il le souhaite et donc ne préciser que la table à atteindre soit, s'il le désire, être très précis en donnant le chemin complet à suivre. Bien entendu, la possibilité lui sera offerte de pouvoir choisir lequel ou lesquels des chemins il désire emprunter et de pouvoir, pour

[2] Une contradiction pourrait être introduite si deux définitions d'un même pointeur le faisaient pointer sur des éléments différents alors que la règle spécifiée dans chaque définition pourrait être équivalente sous certaines conditions.

chacun des chemins et chacune des occurrences de structures rencontrées, accéder aux valeurs de la table de destination.

On remarque donc deux fonctionnalités différentes : la spécification du chemin par l'utilisateur et le travail du programme au niveau des occurrences des structures.

Comme il se peut que l'utilisateur sache que certains chemins ne lui sont d'aucune utilité, il doit pouvoir en exprimer l'idée sur base de ses connaissances; celles-ci ne comportent souvent que des références à des valeurs contenues dans des tables spécifiques. Pour ce faire, la spécification du chemin comportera un argument supplémentaire, mais facultatif, qui permettra à l'utilisateur de préciser par quelles occurrences exactes de certaines tables il désire que le programme passe. Il qualifiera ainsi parfaitement le chemin désiré. Bien entendu, un nombre quelconque[3] d'occurrences pourront satisfaire aux conditions imposées.

Une fois une occurrence de la table de destination atteinte, l'utilisateur peut désirer faire plusieurs choses. De préférence, il devrait à ce moment pouvoir disposer d'un mini langage procédural permettant de gérer les occurrences et les données atteintes. Malheureusement, le prototypage rapide n'est pas possible compte tenu du seul accès en lecture offert pour les 'dumps'. Cependant, lorsque le travail ne se fera plus sur une représentation figée du système d'exploitation telle qu'un 'dump', cette potentialité devra être implémentée dans le programme car le besoin en est bien réel. Ce mini langage devra ainsi permettre d'afficher une table, de manière structurée ou non, de tester certains de ses éléments, d'y enchaîner des actions et enfin de changer d'occurrence.

En globalisant ces nécessités, il devient visible que les procédures quasi automatiques de vérification de la cohérence des structures réelles dans le 'dump' pourront être très facilement écrites en un minimum de temps et par des spécialistes ayant une connaissance même rudimentaire du système d'exploitation en entier, mais connaissant bien sûr parfaitement leur propre sous-système.

---

[3] Ce nombre d'occurrences peut même être nul.

3. L'application : implémentation

Nous allons scinder cette présentation en deux parties, selon les deux résultats demandés par le projet.

3.1. Description formelle des structures de données : le langage d'expression

Selon ce qui a été formulé précédemment dans les spécifications, nous sommes arrivés à la définition des relations[4] suivantes entre structures de données : contient/4, pointeur/5, queue/3 et vecteur/4.

3.1.1. La relation 'contient'

contient (structure contenant, structure contenue, adresse relative en octets du contenu, longueur en octets) : les noms des structures contenant et contenues sont des identifiants. Ces derniers sont généralement ceux repris dans les 'dummy sections'[5] de BS2000. L'adresse identifie l'endroit physique de l'élément décrit comme contenu dans la table, relativement au début de celle-ci et est suivie par la longueur totale de cet élément. Il faut remarquer que la longueur de la table elle-même n'est pas implémentée directement car elle est renseignée dans la table qui la contient ou est calculable par extrapolation du nombre de ses éléments et de leurs longueurs.

Voulant respecter au maximum l'intégrité de la notion de table, nous avons considéré également qu'un pointeur seul, pointant vers une structure, était également une table à lui tout seul. C'est ce que la relation suivante de notre programme implémente :

---

[4] La notation des prédicats utilisés pour la définition des relations utilise la notion d'arité, c'est-à-dire le nombre d'arguments entre parenthèses dans un fait ou une règle Prolog. Ce nombre est toujours noté après le nom du prédicat et précédé du signe /.

[5] Les 'dummy sections' sont des définitions, écrites en un langage structuré proche de l'assembleur, des structures de données présentes dans BS2000. Ces définitions n'ont aucune notion des relations et des types d'éléments formant les structures; elles représentent uniquement le nom et la hiérarchie de toutes les structures.

table (X , X , 0) si pointeur (X , , , ,)
et si pointeur (, X , , ,) .

De plus, afin de permettre une certaine légèreté à la description des structures, nous avons offert la possibilité de ne définir une table que par une relation 'contient', sans obliger à la déclarer d'abord comme table. Vu le nombre de tables du système, cette facilité allège énormément toute description. Voici la règle Prolog qui l'implémente dans notre programme :

table (X , Y , Z) si contient (X , Y , Z ,) .

Ces simples relations et le concept à la fois atomique et non atomique de 'table' permettent de représenter toute architecture de structures de données. Néanmoins, les relations suivantes précisent le type de l'élément contenu.

3.1.2. La relation 'pointeur'

pointeur(nom, nom structure pointée, base, unité, règle) : les noms de pointeurs et de structures pointées sont, bien entendu, identifiants et doivent également faire partie d'une relation contient/4. La valeur du champ 'base' spécifie, si elle n'est pas nulle, l'adresse du segment de base du pointeur, à laquelle l'adresse virtuelle trouvée dans le 'dump' devra donc être additionnée. La valeur du champ 'unité', quant à elle, représente le facteur par lequel l'adresse virtuelle devra être multipliée. Ces connaissances sont rarement exploitées; le plus souvent, elles vaudront respectivement zéro et un. Enfin, le dernier champ comporte la règle dont nous avons déjà présenté le principe. L'implémentation en a été choisie comme suit :

* la valeur un signifie que le pointeur réussit toujours, c'est-à-dire que la valeur du pointeur représente toujours l'adresse de la structure pointée;

* la valeur zéro signifie qu'une valeur de pointeur nulle trouvée dans le 'dump' indique une impossibilité d'atteindre la structure, une valeur positive revenant au premier cas;

* une condition, portant sur des noms identifiants de structures, sera à évaluer mathématiquement et son résultat sera interprété comme au deuxième cas.

Si plusieurs conditions sont nécessaires, plusieurs prédicats pointeurs devront être écrits, chacun comportant une des conditions comme règle. Nous avons déjà présenté les raisons de cette possibilité.

3.1.3. La relation 'queue'

queue(nom, nom des éléments, nom du pointeur pour chaînage) : les noms de la queue et de ses éléments doivent être identifiants. L'élément formant la queue devra au moins faire l'objet d'une définition contient/4 pour son pointeur de liaison, le nom ici fourni ne précisant que le pointeur de liaison sans le définir explicitement. Chaque élément d'une queue peut évidemment comporter n'importe quelle structure. Cependant, les doubles queues devront être implémentées de la manière suivante :

queue (X , Y , pointeuravant)
queue (X , Y , pointeurarrière)
pointeur (pointeuravant , Y , , ,)
pointeur (pointeurarrière , Y , , ,)
pointeursinverses (pointeuravant , pointeurarrière)

On remarque qu'une relation supplémentaire a dû être ajoutée pour que le programme puisse faire la différence entre le pointeur pour chaînage avant et celui pour chaînage arrière.

3.1.4. La relation 'vecteur'

vecteur(nom, nom d'élément, taille d'élément, nombre d'éléments) : la même remarque est encore valable pour les deux identifiants. La taille de chaque élément du vecteur et le nombre maximum de ces éléments présents dans le vecteur permettront au programme d'accéder par un simple calcul à un élément quelconque, connaissant déjà l'adresse du vecteur par une relation 'contient'. Chaque élément devra bien

entendu être lui-même défini par une relation 'contient' s'il n'est pas une structure simplement descriptible, auquel cas une relation 'pointeur', 'queue' ou 'vecteur' suffira.

3.1.5. Un exemple de définition de structures de données

Ce sont là toutes les relations supportées par notre modèle. Le lecteur verra qu'elles sont le plus atomiques possible et que les cohérences nécessaires entre les relations décrites plus haut peuvent facilement être implémentées et vérifiées. Nous pensons qu'un 'shell'[6] aidant à l'écriture de telles relations et les vérifiant serait apprécié par les utilisateurs.

Le lecteur trouvera en annexe le listing du programme comportant, sous le titre 'base de faits de test sur les tables du task management', une description des structures réelles rencontrées dans tout 'dump' généré par BS2000. Sous le titre 'base de faits sur les tables du memory management', il lira une description d'autres structures, description qui n'est pas parfaite mais qui nous a permis de tester tous les cas représentatifs de figures pouvant être décrites dans notre langage. Ces structures de données sont en outre décrites en détail dans [Dedié 88].

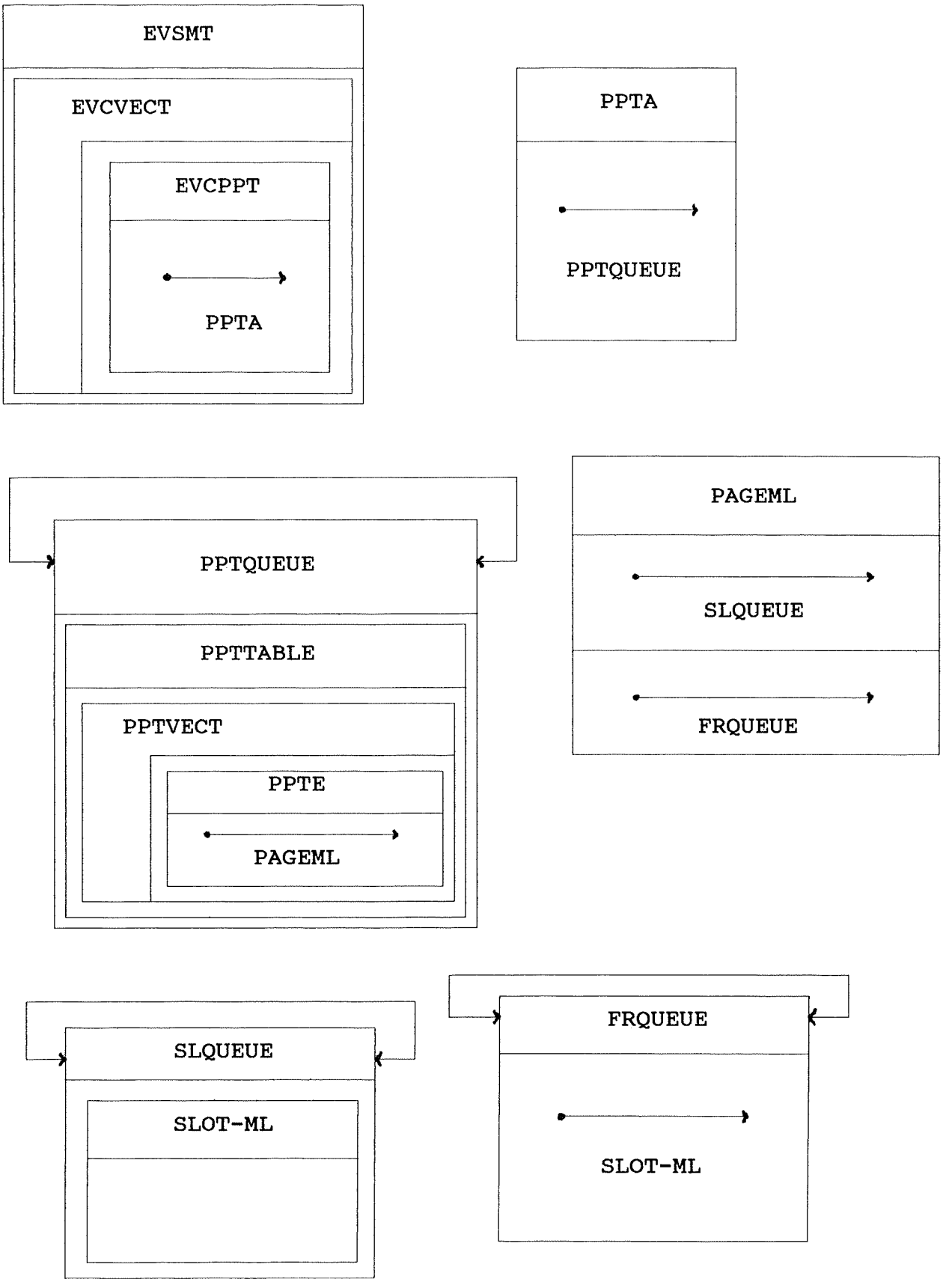
Le lecteur peut notamment y remarquer plusieurs queues imbriquées pouvant amener à des boucles infinies dans lesquelles le programme de parcours, lui au moins, ne se perd pas. Il trouvera également sous le titre 'définition des relations de base' les règles de connaissance[7] sur les règles elles-mêmes de description dont nous avons parlé dans les spécifications.

A titre d'exemple, nous présentons ici, graphiquement, une petite description de structures existantes.

---

[6] Un 'shell' est un programme résolument convivial permettant à un utilisateur de mener à bien une tâche le plus simplement possible.

[7] De telles règles, apportant en fait des connaissances sur les connaissances décrites, sont aussi appelées des méta-règles.


EVSMT
EVCVECT
EVCPPT
PPTA
PPTA
PPTQUEUE
PPTQUEUE
PPTTABLE
PPTVECT
PPTE
PAGEML
PAGEML
SLQUEUE
FRQUEUE
SLQUEUE
SLOT-ML
FRQUEUE
SLOT-ML

(Figure 2.1.)

En langage naturel, ces structures pourraient être décrites comme suit :

la table EVSMT contient le vecteur EVCVECT, formé de tables EVCPPT pointant chacune sur une table PPTA. Toute PPTA pointe sur une queue simple PPTQUEUE. Cette queue contient des tables PPTTABLE contenant chacune un vecteur PPTVECT, ces derniers étant formés de tables PPTE pointant chacune vers une PAGE-ML. Chaque PAGE-ML pointe vers une double queue FRQUEUE et une double queue SLQUEUE. Cette dernière contient toutes les tables SLOT-ML, tandis que la FRQUEUE fait référence à un certain nombre de ces tables.

Cette structure représente une hiérarchie de pages et de morceaux de page[8] pouvant être soit alloués à un processus, soit libres. Les morceaux de page libres sont accédés via la FRQUEUE (pour 'free queue'); tous les morceaux, quel que soit leur état, appartiennent à la SLQUEUE (pour 'slot queue'). Des hiérarchies différentes de pages existent vu les différentes classes de mémoire existant en BS2000.

Voici leur description dans notre langage :

```
contient ( evsmt , evcvect , 0 , 32*4 )
vecteur ( evcvect , evcppt , 4 , 32 )
pointeur ( evcppt , ppta , 0 , 1 , 1 )
queue ( pptqueue , ppttable , linkpointer )
pointeur ( linkpointer , ppttable , 0 , 1 , 1 )
contient ( ppttable , pptvect , 0 , 16*4 )
vecteur ( pptvect , ppte , 4 , 16 )
pointeur ( ppte , page-ml , 0 , 1 , 0 )
contient ( page-ml , ptslqueue , 0 , 4 )
pointeur ( ptslqueue , slqueue , 0 , 1 , 1 )
contient ( page-ml , ptfrqueue , 4 , 4 )
pointeur ( ptfrqueue , frqueue , 0 , 1 , 1 )
queue ( slqueue , slotml , ptavantsl )
```

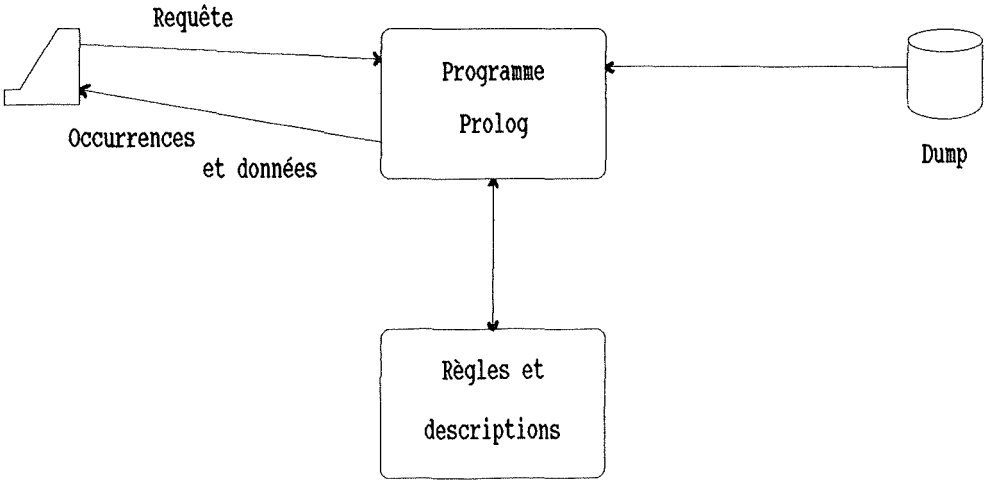
8 Ces morceaux de page sont appelés ' slot ' dans la terminologie des experts en BS2000.

queue (slqueue , slotml , ptarrièresl)
pointeur (ptavantsl , slotml , 0 , 1 , 1)
pointeur (ptarrièresl , slotml , 0 , 1 , 1)
pointeursinverses (ptavantsl , ptarrièresl)
contient (slotml , inconnu , 0 , x)
queue (frqueue , ptslotml , ptavantfr)
queue (frqueue , ptslotml , ptarrièrefr)
pointeur (ptavantfr , ptslotml , 0 , 1 , 1)
pointeur (ptarrièrefr , ptslotml , 0 , 1 , 1)
pointeursinverses (ptavantfr , ptarrièrefr)
pointeur (ptslotml , slotml , 0 , 1 , 1)

Le lecteur pourra remarquer l'adjonction nécessaire de certains éléments, tels PTSLOTML, PTFRQUEUE, PTSLQUEUE et LINKPOINTEUR, n'apparaissant ni dans la description graphique ni dans la description verbale mais nécessaires à la cohérence de notre langage et dont la présence réelle est obligatoire dans la structure physique.

3.2. Le programme de parcours de descriptions sur base d'un 'dump'

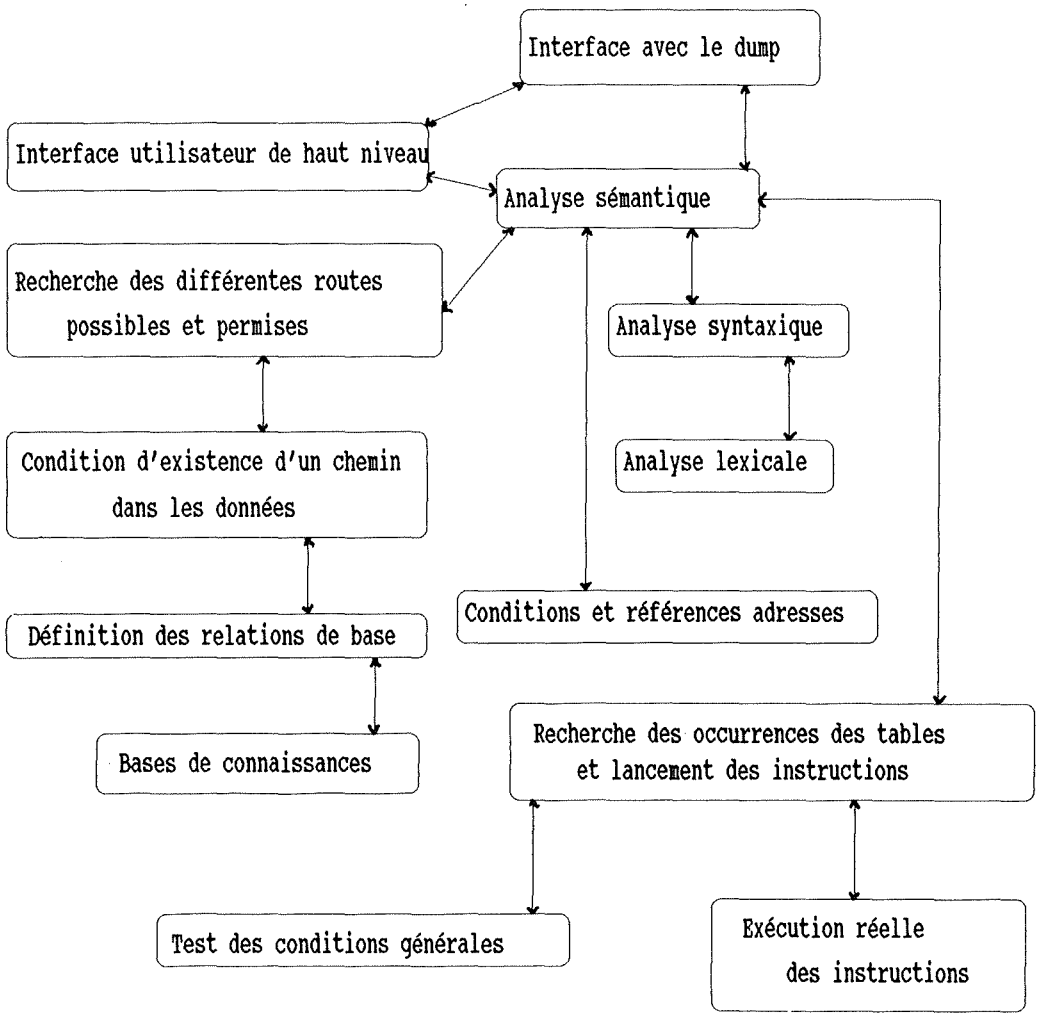
Outre les descriptions de structures, le programme que nous avons construit se compose d'un interpréteur de commandes, d'un parcoureur de descriptions et d'une interface au 'dump'. On peut, globalement, le représenter comme suit :



(Figure 2.2.)

Notre programme recherche, sur base d'une requête compréhensible introduite par l'utilisateur, tous les chemins possibles; pour chacun d'eux il recherche toutes les occurrences respectant les conditions exprimées et il exécute les instructions de l'utilisateur sur les valeurs demandées. Il doit donc, pour rechercher le chemin, utiliser toutes les connaissances sur les structures de données et leurs inter-relations et il doit également, pour fournir les occurrences, pouvoir utiliser le 'dump' physique.

Voici l'architecture logique du prototype :



(figure 2.3.)

Nous allons passer en revue le programme en présentant chacun de ses modules. Lors de chaque présentation, nous essaierons de justifier les choix faits. Quant aux points faibles et aux manquements de notre prototype, ils seront analysés plus en détail au point 4 de ce chapitre. Présentons dans un premier temps les cinq modules centraux de notre architecture :

3.2.1a. L'interface utilisateur de haut niveau

Cette interface a pour but de présenter le programme à l'utilisateur, de lui demander le nom du 'dump' à analyser et de gérer ses différentes requêtes. L'utilisateur peut interrompre le programme entre chaque requête. Elle est très peu sophistiquée et n'existe qu'au titre de point d'entrée obligé du programme.

3.2.2a. L'interface avec le 'dump'

Celle-ci fournit, en plus des primitives d'ouverture et de fermeture, deux primitives d'accès aux 'dumps'. La première primitive offre un 'point d'entrée' dans le 'dump' qui est toujours l'adresse de la table centrale XVT. La seconde lit le 'dump' à une adresse relative à son début et donne en retour la valeur qu'elle y a lue. Comme le lecteur peut le constater, l'accès au 'dump' est très primitif, le 'dump' ne possédant en fait pas de structure propre qui soit d'une quelconque utilité pour notre programme. Etant donné que la seule primitive offerte pour 'commencer' à accéder au 'dump' est la lecture de l'adresse de la table XVT, toute description d'un sous-système quelconque dans notre langage devra être mise en corrélation avec cette table, qui est fort heureusement centrale pour le système d'exploitation. Nous n'avons pu circonvenir cette limitation, imposée par le seul protocole d'accès à des 'dumps' physiques existant, le protocole Anita.

3.2.3a. Les bases de connaissances

Ce ou ces modules de bases de connaissances contiennent, dans le formalisme décrit au point 3.1. , les descriptions formelles des structures de données. Il nous semble important de faire remarquer ici la possibilité intrinsèque de notre programme de disposer d'un nombre quelconque de ces modules. Cette potentialité, nécessaire de par les spécifications mêmes, découle directement du choix du langage d'implémentation Prolog et nous a permis de ne pas avoir à décrire tout le système d'exploitation dans notre prototype. En effet, toute description formelle n'étant qu'une liste de prédicats Prolog, le nombre de ceux-ci n'importe pas. Ainsi, on peut définir autant de tables, de pointeurs, de queues et de vecteurs que l'on veut, et cela indépendamment du programme Prolog lui-même. La seule conséquence possible est un ordre différent de présentation des chemins possibles, chemins qui sont en fait présentés selon l'ordre textuel des prédicats de description formelle. Nous reviendrons sur ce point dans les limites de notre prototype.

3.2.4a. Les définitions des relations de base

Elles implémentent quelques éléments cités dans les spécifications; ces éléments ont été présentés comme implémentant une vue plus logique pour l'utilisateur des relations décrivant les structures de données.

La relation table (nom, nom structure contenue, adresse structure contenue) présente tout ce qui peut, dans la base de connaissances, être considéré comme une table au vu des définitions présentées.

La relation doublequeue (nom) est implémentée en simplification de la structure à cinq prédicats déjà décrite.

Comme le lecteur pourra le remarquer pour la relation table/3, il est très facile de créer des vues spécifiques sur les relations. Le programme est dans ce sens très facilement modifiable pour l'utilisateur, et aussi assez facilement par lui.

3.2.5a. L'analyse sémantique

C'est en fait ce module qui est le coeur du programme. Il utilise différents analyseurs[9] pour avoir une requête précise et correcte de l'utilisateur. Il fait appel à la recherche des chemins possibles dans la description et relève toutes les conditions émises par l'utilisateur puis il parcourt, dans le 'dump', chaque chemin et, pour toute occurrence finale trouvée, il lance les instructions fournies par l'utilisateur. Cette façon de procéder est directement induite des spécifications.

Maintenant que voici présentés les modules principaux de notre architecture, voyons les autres modules, plus concernés par les détails d'implémentation. Cette partie de la présentation est un peu plus technique et nécessite une bonne compréhension de Prolog. Voyons ces modules.

3.2.1b. Condition d'existence d'un chemin dans les données

Outre quelques prédicats de présentation d'un chemin, le coeur de l'algorithme de recherche d'un ou plusieurs chemins dans l'arbre des descriptions est le prédicat unchemin/3. Il se présente comme suit :

```
unchemin ( A , [ A . Finderoute ] , [ A . Finderoute ] ) .
unchemin ( A , [ Y . Finderoute ] , Route ) si
        adjacents ( X , Y ) et
        not membre ( X , Finderoute ) et
        unchemin ( A , [ X , Y . Finderoute ] , Route ) .
```

Comme très souvent en Prolog, ce prédicat est basé sur une récursivité. L'idée de la recherche est la suivante [Bratko 86] : trouver un chemin sans boucle de A à B c'est en trouver un sans boucle de Y à B si Y peut facilement être atteint à partir de A.

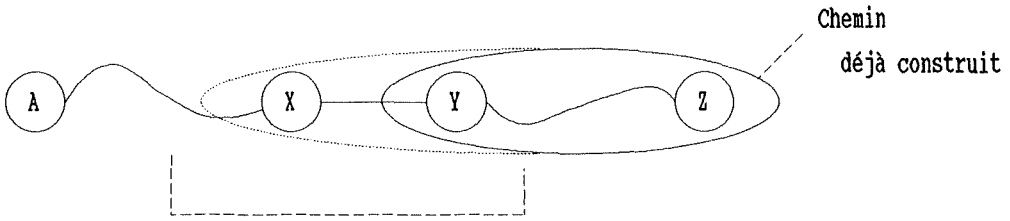
---

[9] Dans notre programme, un analyseur est un morceau de code permettant de décomposer une commande tapée par l'utilisateur en une structure de données; cette structure doit être compréhensible par le programme et doit correspondre à une syntaxe et à une sémantique qui y sont également définies. Dans l'exemple du point 3.2.4b., le lecteur trouvera une implémentation d'un tel analyseur, suivant une grammaire BNF.

On pourra facilement atteindre Y à partir de A s'il existe X directement adjacent à Y tel que X peut être atteint à partir de A. De plus, les noeuds du chemin de A à X ne devront pas appartenir au chemin déjà construit de Y à B. Au départ, l'algorithme commence évidemment avec le deuxième chemin ne comprenant que le noeud cible B, appelé Z dans notre programme. Cela explique la relation d'appel du prédicat unchemin/3 :

chemins (A , Z , Route) si unchemin (A , [Z] , Route).

L'idée de la décomposition en chemin déjà construit et chemin restant à construire peut être représentée par le dessin suivant :



Action d'une itération de l'algorithme

(Figure 2.4.)

Il est également intéressant de remarquer que le noeud Y n'étant jamais précisé et que l'algorithme n'imposant aucune restriction sur lui, Prolog le choisit de ma,ière tout-à-fait aléatoire (ou du moins selon l'ordre des clauses de description). Cela est vrai pour Y, mais pas pour X, puisqu'il doit lui être directement adjacent.

La liberté de choix de Y est cependant très importante. En effet, en cas d'échec de construction d'un chemin de Y à A, cette liberté permet à l'algorithme de faire un autre choix et ainsi d'essayer tous les chemins possibles entre A et Z, bien que cette méthode de recherche puisse se traduire en un nombre exponentiel de noeuds visités. Cet algorithme peut donc aussi être utilisé, en le faisant échouer artificiellement, afin de générer tous les chemins possibles. Il s'agit là d'une caractéristique très importante de Prolog et largement utilisée par notre programme.

3.2.2b. Recherche des différentes routes possibles et permises

Ce module reçoit une table d'origine et une table de destination spécifiées par l'utilisateur et formant les bornes d'un chemin d'accès à une structure bien précise; il vérifie alors si ce chemin est cohérent par rapport aux structures de données décrites dans la base de connaissances. Il est à noter que cette recherche, même si elle se fait sur tous les chemins possibles entre les deux structures fournies, ne renvoit que le premier chemin trouvé. Les autres chemins, en cas de fin de travail sur le premier, sont aussi désirés par l'utilisateur et générés par mise artificielle en échec de la recherche.

3.2.3b. Analyse lexicale

Ce module a pour but de lire une ligne quelconque de caractères à l'écran, d'y assembler les mots et de calculer les nombres représentés par les caractères tapés au terminal. Il fournit ainsi une liste de mots, dont chacun est soit un nom alphanumérique, soit un caractère de séparation, soit un nombre entier. La partie la plus intéressante de ce module est la lecture des caractères et leur concaténation automatique en mots tant que cela est nécessaire. Les deux autres opérations sont réalisées par les prédicats concatséparateurs/2, qui relie par exemple les opérateurs tels < et = en <=, et numériserliste/2, qui remplace toute chaîne de caractères purement numérique par le nombre entier correspondant. Ceci dit, présentons le prédicat lireligne/2, coeur de ce module. Voici son schéma général :

. lire un caractère
. le passer à liremot
. continuer au début, en ayant ajouté le mot construit, jusqu'à ce que
    l'utilisateur termine son entrée.

Bien sûr, la majeure partie du travail est faite par liremot puisqu'il lit les caractères jusqu'à rencontrer un séparateur, assemble ces caractères en un mot et renvoie le mot. Le plus intéressant à noter est l'extrême généralité de ces prédicats, car ils utilisent les prédicats caractère/1 et séparateur/1 pour savoir quels sont les caractères considérés comme des séparateurs ou non. Le lecteur peut aisément voir, dans le listing en annexe, que l'indépendance de ce prédicat par rapport à tout langage

est total, mis à part, bien entendu, le choix des caractères considérés comme des séparateurs par les deux prédicats que nous venons de citer.

3.2.4b. Analyse syntaxique

Notre langage d'interrogation par l'utilisateur peut être défini comme suit par une grammaire BNF[10] :

```
< commande > ::= 'with' < chemin > 'where' < liste conditions > 'do'
                 < liste instructions >
< chemin > ::= < atome prolog > | < atome prolog > '.' < chemin >
< liste instructions > ::= < instruction > | < instruction > ';' < liste instructions >
< instruction > ::= < display > | < list > | 'dialog' | 'failure' | 'nothing' | 'success' |
                    'if' < condition > 'then' < instruction > |
                    'if' < condition > 'then' < instruction > 'else' < instruction >
< display > ::= 'display (' < liste constantes > ')'
< list > ::= 'list(' < liste constantes > ')'
< liste constantes > ::= < constante > | < constante > ',' < liste constantes >
< liste conditions > ::= < condition > | < condition > 'and' < liste conditions >
< condition > ::= < expression > < opérateur de comparaison > < expression >
< opérateur de comparaison > ::= '<' | '>' | '=' | '<>' | '<=' | '>='
< constante > ::= < atome prolog > | < référence >
< référence > ::= < constante > '[' < constante > ']'
< expression > ::= < constante > | < constante > < opérateur arithmétique >
                   < constante >
< opérateur arithmétique > ::= '+' | '-' | '*' | '/'
```

Caractérisons les trois parties de toute ligne de commande (définie dans <commande>) fournie par l'utilisateur :

---

[10] BNF est l'abréviation de Backus Normal Form, du nom de l'inventeur de cette manière de définir univoquement des langages et des grammaires.

With : il est suivi de la spécification d'un chemin. Au minimum, il contient la table de destination, la table d'origine étant supposée être la XVT si aucune table n'est précisée. Mais, implémentation oblige, si une autre table d'origine est spécifiée, le programme commencera toujours son parcours par la XVT puisque c'est la seule table dont il peut directement trouver l'adresse dans le 'dump'.

Un nombre quelconque d'éléments peut être inclus dans le chemin. L'ordre de citation de ces éléments dans la spécification du chemin est bien sûr capital. L'introduction d'une boucle dans une spécification n'est pas autorisée; l'intelligence demandée à un tel outil de parcours supposerait la rétention de trop de points d'orientation intermédiaires, ce qui est tout à fait irréel dans un 'dump' système contenant plusieurs dizaines de milliers de structures inter-reliées.

Where : il permet de spécifier les conditions attendues sur les tables. Si la table à laquelle appartient la référence n'est pas spécifiée, la XVT sera prise par défaut. Il faut remarquer que ce sont de telles conditions qui permettent au programme de rejeter certaines occurrences du chemin trouvé sans en informer l'utilisateur ; l'utilisateur devra donc mesurer très prudemment la portée des conditions qu'il introduira dans cette partie de la commande.

Les conditions peuvent porter sur des éléments n'appartenant pas précisément au chemin spécifié mais appartenant au moins à une table faisant partie du chemin. Tout élément sur lequel porte une condition doit donc soit appartenir au chemin soit être directement contenu (par une relation 'contient') dans un des éléments du chemin.

Do : il offre quelques instructions à effectuer sur l'occurrence de structure de données trouvée comme destination du chemin. Si, le plus souvent, il suffit de présenter une valeur précise à l'utilisateur, il se peut que certaines recherches complémentaires soient désirées. Nous présenterons ces possibilités dans les instructions offertes par notre mini-langage.

Les éléments 'with', 'where' et 'do' forment toute commande. Si nécessaire, l'élément 'where' peut être vide mais ne peut jamais être omis. Voyons maintenant les instructions dont l'utilisateur a besoin pour avoir accès à toutes les informations qu'il désire dans le 'dump'.

L'utilisateur désire avoir la possibilité de changer d'occurrence, c'est-à-dire, dès qu'une certaine occurrence finale a été trouvée et visitée, demander au programme de lui fournir l'occurrence suivante si elle existe. Il doit également avoir la possibilité d'arrêter la recherche s'il sait que toute occurrence supplémentaire lui est inutile. Pour ce faire, il dispose de deux commandes. L'une, 'succeed', permet de dire au programme que tout est fait et qu'il peut arrêter là toutes ses recherches; l'autre, 'failure', dit au programme que le travail sur l'occurrence actuelle est terminé et que l'occurrence suivante est désirée. Il est à remarquer qu'un 'succeed' est toujours implicitement inscrit à la fin du bloc d'instructions 'do'.

L'utilisateur désire aussi pouvoir lister le contenu de la table de destination. L'instruction 'display' effectue cet ordre pour un champ précis de toutes les occurrences trouvées. L'instruction 'display' peut également être appelée 'list'. Initialement, l'instruction 'display' devait être limitée à une seule occurrence, mais nous avons remarqué que son utilisation avec 'failure' donnait en fait un 'list' dans le sens du 'display' actuel. Les deux instructions ont alors été implémentées identiquement.

Ces instructions doivent bien entendu pouvoir être séquentiellement enchaînées et testées. Pour ce faire, outre la séquentialisation, l'utilisateur dispose d'un 'if' à deux ou trois arguments : une condition suivie d'un bloc d'instructions 'then' lui-même suivi, mais pas obligatoirement, d'un bloc d'instructions 'else'; étant donné que les blocs ne sont pas délimitables dans notre langage et pour que les 'if' puissent être emboîtés, l'utilisateur aura recours à l'instruction 'nothing' (qui est une instruction ne faisant rien) pour délimiter les blocs, comme dans l'exemple suivant :

```
if evsmt[XVT] = 84    then if evstt < 2     then display(evsmt)
                                            else nothing
                      else failure
```

ou alors

```
if evsmt[XVT] = 84    then if evstt < 2     then display(evsmt)
                                            else failure
                      else nothing
```

A propos de Prolog, il convient d'ajouter que l'environnement de programmation que nous avons utilisé possédait une syntaxe de base permettant d'inclure directement des clauses BNF ou équivalentes dans un listing Prolog, la conversion étant faite automatiquement lors du chargement du fichier source. Cette possibilité d'écriture se retrouve dans notre listing, dans des cadres faits de '%'. Pour la portabilité de notre programme, nous avons cependant effectué nous-même les transformations nécessaires (peu nombreuses et automatisables) et nous y avons inclus le texte Prolog définitif de notre grammaire. Pour plus de détails sur l'implémentation de telles grammaires en Prolog, nous renvoyons le lecteur à [Shapiro 86].

3.2.5b. Conditions et références adresses

Nous avons déjà vu, au module 3.2.5a, comment l'analyseur sémantique peut chercher tous les chemins possibles. Ayant trouvé un chemin, l'analyseur doit parcourir les conditions et, pour chacune d'elles, rajouter la référence à la table XVT si elle a été omise. Ensuite, le module effectue un travail d'aplatissement de la hiérarchie des conditions de la liste et de complétion référentielle de cette dernière. Ce travail étant fait, l'analyseur sémantique dispose de toutes les données sur le chemin et les conditions exactes requises; il peut donc commencer à interroger directement le 'dump' physique.

3.2.6b. Tests des conditions générales

Ce module, assez simple, permet de tester toute condition émise sur la valeur donnée d'une table et une autre valeur. Les types de conditions permises sont mathématiques, elles sont décrites dans la grammaire BNF du langage de commande.

3.2.7b. Exécution réelle des instructions

Ce module exécute une par une les instructions de l'utilisateur sur l'occurrence trouvée. Il est intéressant de noter ici l'implémentation de la commande 'failure' : un simple échec forcé dans l'exécution du prédicat en cours. Ce dernier oblige le 'backtracking' et, comme le programme est construit pour réussir d'un bloc entre la

découverte d'une occurrence et la dernière exécution d'une instruction, cela force une nouvelle recherche d'une occurrence sans remettre l'exécution des instructions en cause. Ce mécanisme est très différent d'une quelconque structure de contrôle de type procédural et a induit un style de programmation assez particulier mais essentiel pour la facile implémentation de notre projet.

3.2.8b. Recherche des occurrences des tables et lancement des instructions

Comme nous l'avons expliqué, l'analyseur sémantique, après avoir trouvé un chemin et enregistré convenablement les conditions, doit exécuter les instructions sur une occurrence. L'exécution en elle-même a été décrite dans le paragraphe précédent mais c'est ce module-ci qui gère toutes les opérations à faire au niveau des occurrences. Sa tâche est la suivante : pour le chemin trouvé, le suivre en vérifiant les conditions sur les données du 'dump' et exécuter les instructions si une occurrence finale a été trouvée. Pour cela, tout au long de son parcours, ce module doit tenir compte de la nature des objets rencontrés pour le calcul d'adresse dans le 'dump' et, après avoir trouvé les valeurs dans le 'dump', utiliser le module 3.2.6b. pour y vérifier les conditions. Si le module ne trouve pas d'occurrence, il 'backtracke' automatiquement et force la recherche d'un autre chemin, comme cela se passe également lors de l'exécution d'une instruction 'failure'.

Nous avons voulu présenter l'implémentation à un niveau assez logique. Avant de voir tout ce qui pourrait être fait pour améliorer notre prototype, nous dirons un mot des interfaces qu'il a fallu ajouter à notre interpréteur Prolog afin de pouvoir accéder aux 'dumps' physiques.

L'interpréteur Prolog développé par Siemens ne possède pas la possibilité d'appeler directement des routines écrites dans un autre langage de programmation. La seule procédure qu'il offre est la suivante : l'interpréteur étant écrit en langage C, il est possible d'écrire en C des prédicats qui seront considérés comme faisant partie des prédicats de base offerts par l'interpréteur. Cela demande évidemment une recompilation totale de l'interpréteur pour ajouter de nouveaux prédicats.

Nous disposions des procédures d'appel d'Anita, donc d'accès aux 'dumps' physiques, écrites en langage Pascal. Nous avons donc écrit une interface en C, compilée avec l'interpréteur et offrant du côté de Prolog l'appel des procédures écrites en Pascal. Nous ne retiendrons de cela que deux choses : la lourdeur du système et la difficulté de tester l'interface.

4. Les limites du prototype construit

Nous allons structurer ces limites en trois catégories. La première comporte des limites assez simples dont la solution est directement donnée. Bien souvent, il s'agit de détails qui n'ont pas été implémentés faute de temps ou qui ont été oubliés. La deuxième catégorie comporte des limites importantes mais résolubles, requérant seulement plus de temps et/ou de personnes à la tâche. Souvent, nous nous permettrons de proposer des ébauches de solution. Enfin, la troisième partie concerne les limites intrinsèques de notre programme dont une solution exigerait soit la refonte de l'architecture globale du programme soit l'adjonction d'une architecture complète en supplément de celle existante. Cette troisième catégorie comporte entre autres tout ce qui dépasse le cadre d'un prototype ainsi que tout ce que nous avons jugé utile d'ajouter à ce qui nous a été demandé de faire.

4.1. Les détails : première catégorie

4.1.1. Les conditions sur les tables parcourues

Cette limite concerne l'obligation pour l'utilisateur de ne placer des conditions (dans la partie 'where' ou 'do' de sa commande) sur des éléments de tables qu'en référence à des tables qui appartiennent effectivement à tout chemin de parcours retenu. Si ce n'est pas une limite pour la XVT et la table de destination, cela peut l'être pour certaines tables internes au chemin. De plus, cela peut être très ennuyeux lorsque, sur les x chemins corrects, une condition fait référence à une table appartenant seulement à un certain nombre de ces chemins, alors que d'autres chemins ne comportant pas cette table sont corrects. La solution à cette limite est assez directe.

Le module présenté au point 3.2.5b. doit être modifié pour créer les références de toutes les tables, non seulement celles spécifiées dans le 'where' de la commande (ce qu'il fait déjà), mais aussi celles de la partie 'do'.

Mais il y a aussi un autre changement à faire, plus délicat celui-là. Toute condition ayant besoin de l'adresse de l'élément considéré pour être vérifiée, cette adresse n'est mise dans une base de données temporaire que lors du parcours du chemin dans le module 'recherche des occurrences'. Si on veut étendre cette recherche à toute table, il faut que le programme recherche cette nouvelle table et regarde si parmi les chemins y menant un de ceux-ci est bien d'intersection avec le chemin courant. Ce n'est qu'en réalisant cela que l'adresse de l'élément pourra être trouvée sans ambiguïté contextuelle et que le test pourra se faire. Cependant, cette limite relève plutôt de la seconde catégorie puisqu'elle demande une intelligence supplémentaire de la part du programme. Mais elle ne relève pas de la troisième catégorie car son implémentation se fait totalement par appel à des prédicats déjà développés; le problème est le même à la table de destination près.

4.1.2. Non implémentation de la commande 'dialog'

Son implémentation sera la suivante :

```
faire ( dialog ) :-
    lireligne ( L ) ,
    liste-instr ( L , [ ] , Linstr ) ,
    lancer ( Linstr ) .
```

4.1.3. Remonter les erreurs d'Anita

Anita, lorsqu'elle ne peut faire son travail, renvoie un code d'erreur. Ce dernier peut être passé comme argument supplémentaire des quatre prédicats d'accès au 'dump' dans leur implémentation en C. Ce détail est plus une dépendance de la nécessité d'écrire en C des prédicats d'interface avec des éléments hors Prolog qu'un problème du programme lui-même. Certains prédicats, comme liredump/2, remontent déjà des données fournies par Anita et l'implémentation ne pose aucun problème.

4.1.4. Parfaire la généralité des conditions

Dans l'état actuel, notre programme ne vérifie que des conditions du type exvtlta[XVT]>24, et non exvtlta[xvt]>8*3. Cette extension est très rapidement implémentée par la même technique d'application d'opérateurs qui est utilisée pour appliquer, dans ces mêmes conditions, les opérateurs de comparaison. Cette technique tournant parfaitement bien, l'extension aux opérateurs de calcul est immédiate.

4.1.5. Cas de plusieurs conditions dans le 'where'

Plusieurs conditions peuvent y être signifiées, séparées par 'and'. Le programme, dans sa forme actuelle, et même s'il y a plusieurs conditions, ne vérifie que la première des conditions. Etant construit pour les vérifier toutes, il s'agit d'une omission accidentelle de notre part.

4.1.6. Explication d'échec dans le parcours d'un chemin choisi

Lorsque toutes les conditions ne sont pas respectées lors du parcours du chemin, le programme en choisit automatiquement un autre. Cela pourrait être plus raffiné. En effet, au moment de l'échec, le programme pourrait présenter le chemin actuel et la condition non vérifiée, ce qui aiderait l'utilisateur dans la vérification de l'adéquation de ses conditions. Cela existe déjà comme outil de test où toutes les conditions et tous les chemins sont imprimés. De telles impressions sélectives rendront le programme plus agréable d'emploi.

4.1.7. 'List', 'display' et 'if'

Le 'display', comme nous l'avons déjà signalé, agit en fait comme un 'list'. Il s'agit là d'un détail révélé de la réalisation du programme et qui a montré

l'incohérence de la différence entre les deux commandes. Ce cas ayant déjà été réglé, il nous reste celui du 'if' pour lequel, en le jumelant avec l'instruction 'nothing' pour gérer les conditions emboîtées, nous avions trouvé une solution assez boîteuse . L'idéal, bien sûr, est d'ajouter à notre mini-langage la notion de bloc d'instructions délimité par BEGIN et END, empêchant ainsi toute ambiguïté. Pour un emboîtement de conditions simples, une option devrait également être prise par défaut, comme dans tout langage procédural.

4.1.8. Parcours des queues

Nous avons remarqué, dans le prédicat vériflien/4 de parcourirchemin, que les queues ne sont pas implémentées. Cet oubli est passé au travers de notre base de tests puisqu'elle ne contenait pas de queue ! Mais l'implémentation est facile et identique aux autres structures.

4.1.9. Lourdeur des doubles queues

Les doubles queues sont actuellement définies par une combinaison de trois prédicats différents dont deux sont utilisés deux fois dans notre langage de structuration. Ces cinq lignes ne pourraient-elles pas être ramenées à deux comme pour les queues ? Si, bien sûr, en définissant un nouveau prédicat doublequeue et en augmentant les détails requis par pointeursinverses/2. Notre choix de définition des doubles queues par rapport aux simples queues nous apparut logiquement cohérent par rapport à la spécification, car si une queue peut être parcourue dans les deux sens, notre prototype, de par son rôle de pure consultation, n'aurait jamais utilisé le sens arrière. On peut donc laisser telles les définitions mais il faut permettre à l'utilisateur d'user d'un prédicat doublequeue/1 de définition qui serait alors automatiquement traduit par notre programme d'exploitation dans la forme actuelle d'une double queue.

4.2. Les manquements : deuxième catégorie

4.2.1. Présentation graphique plus sophistiquée de l'interface utilisateur

Nous voulons parler d'une amélioration de l'interface utilisateur actuelle, ce qui n'est d'ailleurs pas très difficile à réaliser vu son état. Dans cette version, l'interface se borne à demander des commandes à l'utilisateur, à lui fournir les valeurs qu'il demande au niveau des occurrences des tables spécifiées et à révéler de temps en temps les fonctionnements fautifs et leur raison.

Bien que tout cela soit nécessaire, il serait très intéressant de pouvoir disposer d'une interface utilisateur purement graphique qui serait alors à même de présenter à l'utilisateur, pendant la recherche d'occurrences aussi bien que lors des moments d'interaction avec lui, les structures rencontrées et visitées, dans un formalisme du genre de celui que nous avons utilisé pour présenter un de nos exemples plus haut.

Une telle interface, incluant l'utilisation de fenêtres graphiques et de menus, peut très bien être écrite en Prolog. Le lecteur trouvera dans [Hepburn87] des modules utilitaires gérant menus déroulants et fenêtres.

4.2.2. Indépendance du choix du chemin par rapport à la représentation en Prolog (plus possibilité d'y mettre des boucles pour arriver à certaines conditions)

La découverte, par notre programme, des chemins répondant aux spécifications de l'utilisateur se fait nécessairement dans un certain ordre; en l'occurrence, il s'agira de l'ordre des faits en Prolog. Contrevenir à cette limitation dans l'ordre de découverte des chemins n'est possible que d'une seule manière : rechercher d'abord tous les chemins possibles puis les présenter selon un ordre choisi. Cet ordre peut alors faire partie de la base de connaissances elle-même ou être un choix éclairé fait par l'utilisateur.

4.2.3. Enrichissement des messages d'erreur lors de l'analyse syntaxique

Cette limitation est nettement plus difficile à maîtriser que la précédente. Nous n'avons pas trouvé d'exemples dans la littérature qui nous permette d'y arriver. La philosophie de tels messages impliquant une 'dé-généralisation' de l'analyseur employé, nous n'avons pas jugé utile de nous y attarder. Cependant, l'analyseur, répondant en

détail à toutes les erreurs, devra être nettement plus puissant que le nôtre étant donné l'obligation de prévoir tous les cas d'erreur possibles. Remarquons que le principe du simple échec d'une partie de l'analyseur sur une erreur n'est pas possible ici parce que ce dernier utilise déjà ce mécanisme comme base de sa recherche de toutes les interprétations possibles, limitées bien sûr à une seule en fin de compte dans notre programme.

4.2.4. Possibilité de partir d'une autre table que la XVT

Par défaut, qu'elle soit spécifiée dans le chemin ou non, cette table est actuellement toujours considérée comme le premier élément de tout chemin de recherche. Cette règle ne pourra pas changer, en raison des méthodes d'accès aux 'dumps'. Cependant, il serait très intéressant pour l'utilisateur de pouvoir commencer son chemin à partir d'une autre table. La XVT sera alors ajoutée directement en tête du chemin spécifié. C'est aussi le cas actuellement.

Pourquoi alors spécifier un besoin particulier ? Pour la raison suivante : l'ajout 'artificiel' que le programme fait actuellement ne permet pas à l'utilisateur de choisir son chemin entre la XVT et la table qu'il a spécifiée comme début de son propre chemin. On voit ainsi que l'absence de départ de la XVT chez l'utilisateur entraîne des choix que le programme devrait soumettre à l'utilisateur alors que, pour le moment, il se contente de décider lui-même du chemin à choisir; celui-ci est en l'occurrence le premier selon l'ordre des prédicats Prolog, comme nous l'avons déjà précisé.

L'inconvénient de ne pouvoir choisir son chemin entre la XVT et sa propre table de début pourrait en partie être levé par l'adjonction au prototype de la notion de point d'entrée dans un sous-système. Ce point d'entrée serait une table qui est considérée comme centrale pour une partie bien déterminée du système d'exploitation. En général, cette table regroupera les fonctionnalités de base propres au sous-système qu'elle supporte. Le système de recherche de chemins pourrait alors comprendre des prédicats du genre cheminarchivé/3, qui comprendrait une table de départ, le plus souvent la XVT, une table d'arrivée, le point d'entrée et le chemin entre les deux. Il serait bien entendu nécessaire que l'utilisateur spécifie lui-même ce chemin selon le travail qu'il désire faire. Mais comme ce chemin est souvent identique pour le développeur d'un sous-système, ce mécanisme lui permettra de spécifier uniquement son point d'entrée comme origine du chemin, le prototype ajoutant alors automatiquement à chaque

chemin trouvé celui qui mène de la XVT au point d'entrée. Il est important de noter qu'un tel système ferait également gagner beaucoup de temps de recherche à Prolog, le chemin de la XVT au point d'entrée étant très souvent le plus long.

4.3. Au-delà d'un prototype : troisième catégorie

4.3.1. Possibilités de prototypage rapide

Par prototypage rapide, nous entendons la possibilité offerte à l'utilisateur de pouvoir agir en écriture sur le 'dump' physique. Si notre langage contenait des instructions d'écriture de valeurs dans le 'dump', en plus des lectures actuellement présentes, il serait alors possible, par exemple, de changer automatiquement toute référence 'pointeur' à une certaine table par un pointeur vers une autre table, ceci est vrai pour toutes les tables et toutes leurs occurrences dans le 'dump'. Cela implique, comme on peut le voir, une vision quelque peu différente d'un 'dump' : il serait alors vu comme une photo d'un système, susceptible de subir des modifications, et capable d'être 'réinjecté' dans la machine; le système d'exploitation aurait ainsi la possibilité d'être relancé afin de montrer l'impact des changements effectués.

Outre la détection des erreurs survenues dans un système, une telle vision d'un 'dump' permettrait la correction de ces erreurs et même la vérification de la validité des corrections effectuées. Mais il s'agit là d'une application très précise de notre prototype qui ne pourrait être effectuée que par des spécialistes du système d'exploitation. Elle se montre cependant très intéressante car elle offrirait une aide dans une tâche qui, jusqu'à présent, se fait presque entièrement manuellement.

4.3.2. Récupération des 'dummy sections' de BS2000

Cette idée permettrait une automatisation complète du langage de spécification des structures de données qui a été décrit. Le système BS2000 comporte déjà des descriptions de ses structures. Bien que ne comportant que des informations d'un niveau sémantique moins riche que celui de notre langage, ces structures pourraient alors être automatiquement traduites par un programme dans notre langage de description. Ce programme supplémentaire pourrait alors être vu comme une

possibilité de notre prototype d'enrichir sa base de connaissances sans effort humain de spécification. Cette possibilité nous avait été demandée par Siemens, mais, par manque de temps, nous n'avons pu nous y atteler.

4.3.3. Shell d'écriture et de vérification de cohérence des descriptions

Faisant suite à l'idée précédente, il nous apparaît également intéressant de disposer d'une fonctionnalité de vérification des descriptions fournies au prototype. Ces descriptions, qu'elles soient écrites par des utilisateurs ou par le programme de conversion des *dummy sections* de BS2000, pourraient ainsi être vérifiées sous l'angle de la cohérence du modèle qu'elles décrivent. Les descriptions contenues dans la base de connaissances forment un modèle d'entités de type unique, c'est-à-dire des tables, et de relations, pouvant être des pointeurs, vecteurs, etc.

Bien qu'un tel modèle ne soit pas du type précis de ceux que l'on peut rencontrer lors de la définition de bases de données, il s'en rapproche fort. Cela peut également être mis en évidence par le caractère très proche du modèle relationnel que possède notre implémentation des descriptions en Prolog. Il nous apparaît donc tout-à-fait raisonnable d'offrir la possibilité de vérifier la cohérence de ces descriptions.

En plus de sa fonctionnalité propre de vérification, cet outil pourrait bien entendu permettre à l'utilisateur d'écrire ses descriptions à l'aide d'un éditeur syntaxique et même de les décrire graphiquement. Cette possibilité peut être approchée par une description des problèmes rencontrés en warnings, qui sont des descriptions pouvant mener le prototype à agir d'une manière contraire à la volonté du descripteur, et en errors, qui sont des descriptions incompréhensibles par le prototype. Nous noterons que l'emploi d'un éditeur syntaxique permettra déjà d'éliminer toute erreur syntaxique et aussi de décharger l'utilisateur de la syntaxe propre à Prolog.

Présentons maintenant la petite typologie que nous avons pu faire ressortir de notre connaissance du prototype :

4.3.3.1. Les 'warnings'

Les déclarations suivantes devraient générer un avertissement lors de leur vérification :

- des tables qui sont déclarées deux fois, car cela induirait Prolog à ne jamais considérer que la première table dans l'ordre de description, bien que la sémantique d'une telle déclaration puisse être valable. Mais une définition multiple serait toujours permise pour les pointeurs.

- des pointeurs inversibles auxquels aucun inverse n'aurait été défini, l'éditeur syntaxique ne pouvant imposer la définition d'un inverse chaque fois que cela est possible.

4.3.3.2. Les 'errors'

Les déclarations suivantes devraient automatiquement générer une erreur lors de leur vérification :

- des tables inaccessibles, c'est-à-dire des tables qui ne seraient l'objet désigné d'aucune relation.

- l'incohérence de définition d'une double queue, pouvant survenir lorsque les règles des pointeurs définis comme inverses ne sont pas compatibles.

- des tables référencées par des éléments non déclarés, telles des tables pointées par un pointeur qui ne serait pas défini comme objet d'une quelconque autre table.

Chapitre 3 : Au-delà de Prolog.

Introduction.

Ce chapitre pourra paraître quelque peu hybride au lecteur car il comprend deux choses bien différentes. Après avoir présenté au chapitre 1 les fondements théoriques que nous avons utilisés dans notre application, et après avoir détaillé cette application au chapitre 2, nous désirons développer deux autres manières d'aborder le type de problème que nous avons traité.

La première approche, présentée au point 1, sera celle des logiques ambiguës, dites aussi logiques révisables, qui permettent de formaliser des connaissances incertaines. Bien qu'aucun langage de programmation ne les implémente à l'heure actuelle, beaucoup ont été utilisées et formalisées dans des systèmes experts. Leur usage est souvent indispensable pour le traitement de certains types de problèmes comme le diagnostic ou le raisonnement sur des connaissances incomplètes.

La deuxième approche, présentée au point 2, est celle de Prolog II, extension de Prolog, inventée également par A. Colmerauer. Cette deuxième version du langage Prolog apporte une philosophie plus complète qui nous a paru intéressante.

1. Les autres logiques.

L'introduction à la théorie de la logique que nous avons faite au premier chapitre était centrée sur la représentation des connaissances. La logique peut également servir de formalisme de référence et de moyen d'analyse sémantique de la connaissance représentée. Les représentations réseau et objet présentées au chapitre 1 ont également montré que tout en étant des mécanismes de représentation de connaissances, elles possèdent un pouvoir d'expression supérieur à celui de la logique, bien que leurs descriptions puissent dans un certaine mesure être traduites en représentations logiques.

Des logiques plus puissantes que la logique des prédicats existent et dépassent en pouvoir d'expression cette dernière, ainsi que les représentations objet et réseau. Les paragraphes suivants présenteront quelques-unes de ces logiques qui se focalisent surtout sur les caractères spécifiques d'expressivité qu'elles possèdent. Cette

présentation est un résumé du chapitre 4 de [Thayse 89], enrichi de quelques éléments fournis par [Turner 84].

1.1. La logique et le raisonnement révisable.

Un système logique voulant formaliser des déductions révisables, ou des déductions sur des connaissances incomplètes, ne peut être que non-monotone. En effet, le nombre de théorèmes qu'un tel système peut décrire doit pouvoir aussi décroître lors d'une augmentation du nombre d'axiomes de base.

Etant par définition monotone, la logique classique n'est pas capable de formaliser un raisonnement incertain. Quant à la construction d'une logique non monotone, elle nécessite la définition d'une relation d'inférence permettant de tirer des conclusions qui pourraient ne pas être vraies dans tous les modèles. Ces formules déduites sont appelées plausibles.

L'acceptation d'une formule comme plausible n'est adéquate que si cette formule est vérifiée dans au moins un des modèles des formules qui l'ont prouvée. On exige également que des formules plausibles mais inconsistantes entre elles ne soient jamais inférées conjointement. Cette exigence de constance apporte la possibilité de révisabilité d'un raisonnement.

L'adjonction de formules plausibles à un système de déductions nécessite de sa part la possibilité d'accepter des conclusions qui diffèrent selon l'ordre dans lequel elles ont été tirées. Les conclusions possibles ne pourront plus être énumérées, puisqu'elles pourront être inférées puis rétractées. La notion de théorème se présente alors comme une formule valide dans tous les ensembles de formules inférées stables et une démonstration devient l'établissement de l'existence d'un tel ensemble stable. Bien entendu, sans la propriété de monotonie, la plupart des métathéorèmes de la logique classique ne sont plus valables.

Pour assurer par lui-même la rétractabilité de ses inférences, un système non monotone doit contenir des règles d'inférence révisables. De telles inférences agissent alors dynamiquement et infèrent des formules tant qu'elles ne peuvent pas inférer leur contraire selon la logique classique. Mais pour tester l'inférabilité d'une formule, on

doit utiliser toutes les formules déjà inférées, dont les formules plausibles; il y a alors danger de circularité dans l'inférabilité.

Pour combattre ce danger et pour définir des mécanismes d'inférence, plusieurs systèmes logiques ont été inventés; en effet, par essence, tous les raisonnements révisables ne sont pas de la même nature. Les points suivants présenteront quelques unes de ces théories. Nous avons essayé que cette présentation de logiques différentes, bien que non exhaustive, soit assez représentative des idées que le lecteur pourra trouver dans la littérature.

1.2. Les logiques des défauts.

Elles ont été développées pour formaliser le raisonnement simplement consistant, c'est-à-dire le raisonnement qui fait admettre comme générales des formules qui ne le sont pas vraiment mais qui n'admettent que peu d'exceptions. C'est ainsi que raisonne d'ailleurs le plus souvent l'esprit humain.

Ce raisonnement par défaut est implémenté selon des règles particulières appelées 'défauts' ayant la forme suivante :

$$\frac{a : M\ b}{c}$$

La signification est celle-ci : si a est cru et si b est consistant avec tout ce qui est cru, alors c peut être cru également. On peut par exemple énoncer que, en règle générale, tous les oiseaux volent par :

$$\frac{Oiseau(x) : M\ Vole(x)}{Vole(x)}$$

Cette règle de défaut permet de traiter les cas d'exception sans nécessiter l'identification préalable de ces cas. Une théorie comportera donc tout un ensemble de tautologies, de formules vraies, ... avec en plus des règles de défaut. Ces règles définissent donc en plus des ensembles consistants de croyances. Ces ensembles sont aussi appelés extensions de la théorie avec défauts.

Une classe particulière de théories avec défauts est celle des théories normales. Pour elles, le conséquent et la justification d'un même défaut sont identiques. Un défaut normal sera donc par définition de la forme suivante :

$$\frac{a(x) : M\ b(x)}{b(x)}$$

Ces théories normales admettent au moins une extension et possèdent la propriété de semi-monotonie. Cette dernière assure qu'une augmentation du nombre de défauts crée une nouvelle théorie admettant une extension comprenant l'ancienne extension. La conséquence pratique la plus importante de cette propriété est "la possibilité de construire une théorie de la preuve qui reste locale par rapport aux défauts mis en jeu" [Thayse 88].

1.3. Les logiques modales de connaissance et de croyance.

Le but premier de ces logiques est la formalisation des concepts de nécessité et de possibilité. La modélisation et l'analyse des paradigmes de connaissance et de croyance forment également un des champs d'application des logiques modales. Ces deux concepts seront implémentés sous la forme d'opérateurs dits 'modaux'. Présentons les principes de base des logiques modales, logiques qui sont toujours des extensions de la logique de premier ordre.

Un langage modal contient deux opérateurs, notés L et M. L'opérateur L prend la signification 'est cru' dans les logiques de croyance et 'est connu' dans les logiques de connaissance. Quant à l'opérateur M, son dual, il prend respectivement les significations 'l'inverse n'est pas cru' et 'l'inverse n'est pas connu'.

Tout langage modal supporte tous les théorèmes de la logique des propositions; il connaît en plus la règle du modus ponens, le schéma d'axiome de distribution qui s'écrit $L(p=>q)=>(Lp=>Lq)$ et se lit ' s'il est nécessaire que p entraîne q, alors il est nécessaire que p entraîne il est nécessaire que q ', et il connaît également la règle d'inférence modale de nécessité disant que ' p est nécessairement vrai si p est vrai '.

La modalité apporte ainsi un surcroît d'expressivité à la logique classique. Pour définir une sémantique formelle de ces logiques améliorées, le concept de monde possible a été introduit. Ce concept permet de vérifier toute formule modale par rapport à un certain monde de l'univers. Un tel monde peut être complètement bouleversé pas la simple assertion d'un fait, d'où le qualificatif de possible. Mais si un monde contient une sémantique pour les formules, les mondes possibles sont reliés entre eux par une relation d'accessibilité qui indique la succession des différents moments auxquels le monde est considéré.

Des systèmes modaux plus élaborés peuvent être obtenus par ajout d'axiomes au système modal normal. Le schéma d'axiome de la connaissance ajoute l'axiome 'p=>Lp' au système modal, l'obligeant à ce que ' ce qui est connu soit vrai '. Un deuxième axiome, le schéma d'axiome de l'introspection positive, noté 'LLp=>Lp' affirme que ' si je connais p, alors je sais que je connais p ' et est nécessaire pour formaliser une intelligence introspective parfaite. Enfin, un troisième de ces axiomes est le schéma d'axiome de l'introspection négative. Il s'écrit LMp => Mp et signifie ' si je ne sais pas que p est vérifié, alors je sais que je ne sais pas que p est vérifié'. Cette propriété, inintéressante à première vue, est toutefois très exigeante. Elle exprime en effet la parfaite connaissance des limites de la connaissance dans le système.

Un système modal regroupant les trois axiomes que nous venons d'évoquer est nécessaire à la caractérisation de la connaissance d'un agent intelligent ayant une parfaite capacité d'introspection logique sur ce qu'il connaît et sur ce qu'il ne connaît pas. Les logiques non monotones de McDermott forment un tel système.

1.4. Les logiques non monotones de McDermott et Doyle.

Ces logiques s'écartent des logiques des défauts sur trois points principaux : le cadre logique qu'elles ont choisi est celui des systèmes modaux de possibilité et de nécessité; leur système est un système axiomatique universel car il ne formalise pas un ensemble de règles non monotones propres au domaine de l'application; l'intérêt se porte non sur une extension particulière d'une théorie mais sur les formules présentes dans toutes les extensions d'une théorie.

Les auteurs de cette logique ont défini une technique permettant d'éviter la circularité dans la définition des règles d'inférence non monotones, danger dont nous avons déjà parlé. Basé sur la solution à une équation d'un point fixe[1], le système axiomatique solution peut être vu comme un système modal classique augmenté d'une règle particulière permettant d'inférer des assertions consistantes.

Le point fort de la logique non monotone de McDermott est l'utilisation de la théorie du point fixe pour caractériser les ensembles stables de conclusions d'un système non monotone ainsi que l'usage de la logique modale pour formaliser le raisonnement révisable. Lors de l'établissement de sa théorie, McDermott a montré que le choix du système modal le mieux adapté restait problématique. Comme exemple d'un choix judicieux permettant à une théorie logique de modéliser un agent idéalement rationnel qui raisonne de façon introspective sur un ensemble initial de croyances, présentons les logiques autoépistémiques.

1.5. Les logiques autoépistémiques.

Ces logiques ont pour objet la formalisation d'un raisonnement introspectif et idéalement rationnel, opéré sur un ensemble initial de croyances. Elles permettent notamment des raisonnements de la forme ' si je ne crois pas que p est vérifiée, alors q est vérifiée '. Comme nous l'avons vu, les logiques modales de connaissance et de croyance sont bien adaptées à la formalisation de ce type de raisonnement.

Les logiques autoépistémiques peuvent être vues comme une reconstruction des logiques non monotones de McDermott dans lesquelles les paradigmes d'inférabilité et de consistance sont remplacés par la formalisation de certaines capacités introspectives de raisonnement. Ainsi, les opérateurs modaux M et L, au lieu de formaliser le consistant et l'inférable, formalisent respectivement ' l'inverse n'est pas cru ' et ' est cru '. Ces logiques autoépistémiques ont aussi la même expressivité formelle que les logiques des défauts.

Une caractérisation sémantique intéressante des logiques autoépistémiques est basée sur la notion de mondes possibles. Elle permet de démontrer l'existence de théories autoépistémiques complètes et légales vis-à-vis d'un ensemble de prémisses.

---

[1] Une équation de point fixe porte sur la relation d'inférabilité définie par le système non monotone.

Cette possibilité permet notamment, sur base d'un certain nombre de croyances, de générer tous les mondes possibles dans lesquels toute formule est vérifiée. Cette caractéristique permet donc de projeter tous les mondes cohérents avec un nombre de prémisses, mondes pouvant être très différents selon les croyances acceptées.

La logique autoépistémique permet ainsi de caractériser les conclusions que l'on attend d'un système ayant un pouvoir d'introspection parfait. Cette aptitude peut être très utile si l'on veut interroger une base de connaissances sur ses propres limites de connaissances.

2. Prolog II.

La présentation de Prolog II est faite sur base des éléments supplémentaires qu'il apporte à Prolog et que nous avons trouvés dans [GKPC 86]. Une mention est également faite à ces nouvelles possibilités au chapitre 9 de [Kluzniak 85]. Voici ces éléments supplémentaires.

2.1. Les arbres.

Prolog connaît la notion d'arbre et d'égalité entre arbres finis. Prolog II ajoute la notion de contrainte : un système d'égalités et d'inégalités entre plusieurs arbres pouvant, par exemple, implémenter directement un système d'équations. Prolog II possède un algorithme puissant pour résoudre ce problème particulier : prenant le système d'arbres, l'algorithme réduit non déterministiquement les contraintes d'unification entre les différents éléments des arbres jusqu'à trouver un système d'unifications accepté par tous les arbres. La recherche peut bien entendu être relancée pour trouver des solutions supplémentaires car elle n'est pas déterministe et peut être incluse directement dans un schéma de programmation fonctionnant par réussite/echec, schéma de base de Prolog.

Cette particularité de traitement des arbres est étendue aux listes qui sont considérées comme des cas particuliers d'arbres. Ainsi, avec plusieurs listes de listes,

il est possible d'utiliser les mécanismes décrits au paragraphe précédent, ce qui n'était pas possible en Prolog.

Une structure d'arbre est cependant particulière à Prolog II : les arbres infinis. Un arbre est infini quant au moins une de ses branches l'est. Un arbre infini aura toujours un noeud faisant directement référence à un noeud précédent de la même branche. A condition de prévenir Prolog II par le prédicat 'infinite' que certains arbres peuvent être infinis, le moteur d'inférence travaille indifféremment avec des arbres finis ou infinis.

2.2. Négation et différence.

Prolog contient un prédicat, appelé equal/2, testant l'égalité de deux termes quelconques[2], et un opérateur de négation, malheureusement déterministe. Prolog II apporte un prédicat supplémentaire qui ne remplace pas la négation mais qui permet de tester de manière non déterministe la non égalité de deux termes quelconques. Le prédicat est dif/2. Ce prédicat teste la différence des termes comparés quel que soit leur type, mais à condition que les types soient compatibles. Ainsi, il peut tester la différence entre deux arbres ou entre une liste et un arbre.

Le caractère non-déterministe de dif/2 est très important. Si, lors du déroulement d'un programme, la différence a été testée et prouvée mais que plus loin l'exécution échoue et backtracke, dif/2 générera de nouvelles instantiations vérifiant la différence des deux termes. Dif/2 est donc une implémentation totalement logique de l'opérateur composé ' ~= '.

Malheureusement, la négation obéit toujours au même principe de non déterminisme aussi bien en Prolog II qu'en Prolog. Mais Prolog II assure qu'une double négation sera toujours traitée comme l'absence de négation, ce qui n'était pas toujours le cas de Prolog; cette limitation d'implémentation n'était d'ailleurs pas documentée et elle nous a réservé quelques surprises lors du développement de notre application.

---

[2] Pour ce faire, Prolog essaye en fait d'unifier les deux termes.

2.3. Les blocs.

Le prédicat block/2 permet de définir un bloc virtuel autour d'un prédicat. Il est alors possible de faire réussir artificiellement l'exécution du prédicat avec l'instruction block-exit/1. Cette dernière primitive génère un message d'erreur définissable par l'utilisateur mais n'interfère pas avec le comportement du moteur d'inférences. Ces deux prédicats offrent une gestion d'erreurs dans Prolog II qui n'était pas supportée par Prolog.

Les erreurs pouvant être définies par l'utilisateur, ce dernier a la possibilité de redéfinir certaines erreurs du système et ainsi empêcher par exemple l'arrêt du programme par la frappe de la séquence de caractères < Control - C >.

2.4. Le raisonnement sur des données incomplètes.

Il est parfois utile de ne prendre une décision ou de ne lancer un raisonnement que lorsque l'on a certaines données à sa disposition. Cette possibilité est offerte par le prédicat freeze/2. Son premier argument est une variable et son deuxième un prédicat. Freeze/2 ne lancera l'exécution du prédicat que lorsque la variable aura été instantiée.

Bien sûr, ce prédicat détruit l'aspect déclaratif du langage mais il permet d'appliquer le mécanisme de recherche de solutions, propre à Prolog, sur des connaissances incomplètes. Ce prédicat peut cependant mener à certaines erreurs, car Prolog II ne s'assure jamais qu'une variable gelée a été instantiée et a bien déclenché l'évaluation du prédicat correspondant.

2.5. Test de type.

Prolog II offre des prédicats pour vérifier si un terme est la représentation d'un réel, s'il est une liste ou s'il est un t-uple. Cela n'existait pas en Prolog. Nous avons fait l'expérience de ce défaut de possibiliter de tester le type liste dans notre application, où nous devons essayer d'appliquer un prédicat renvoyant le premier

élément d'une liste au terme dont nous devons tester le type. Seul unn échec de ce prédicat nous assure que le terme n'est pas une liste.

2.6. La notion de monde.

L'ensemble de règles de toute application est organisé en mondes; cette notion permet une organisation modulaire pour de très gros programmes. Prolog II contient au moins trois mondes : le monde 'superviseur', qui contient tous les appels aux prédicats prédéfinis, le monde 'base' qui contient tous les accès à ces prédicats et qui permet d'isoler les primitives du superviseur de celles de l'utilisateur, et le monde 'normal' dans lequel se trouve l'utilisateur.

Dans un monde existe un prédicat courant. Des primitives permettent de changer ce prédicat courant sur base d'un numéro ou d'une qualification du prédicat. On peut également ajouter ou supprimer dynamiquement des règles, mais cette possibilité existait déjà en Prolog.

Le choix du monde courant se fait grâce à des primitives telles que 'down' et 'climb' qui permettent de voyager dans l'arborescence des mondes. Bien entendu, on peut créer et effacer des mondes. Mais une fois situé dans un monde, l'utilisateur n'a accès qu'aux règles définies dans ce monde et dans ses ancêtres.

Conclusion.

Le but premier de notre prototype est de rencontrer les désirs exprimés par la firme Siemens. Dans l'état actuel de son développement, il a servi de démonstration au bien-fondé de notre approche pour ce type de problème. Dans quelque temps, Siemens envisage de développer notre prototype à Munich pour en faire un outil de production, sans doute en l'intégrant à un outil déjà existant.

Nous espérons avoir montré que les avantages offerts par Prolog sont très importants quand il s'agit de développer un prototype et de justifier une approche. Ces avantages sont allés de délais de développement très réduits à des tests très simplifiés et à un programme très rapidement exécutable pour des démonstrations. Cet avantage n'est pas à négliger dans la relation avec le client du logiciel construit.

Le problème de la recherche d'un chemin dans une structure et celui de la définition d'une base de connaissances pour exprimer ces structures, de fait les deux problèmes de base de notre prototype, étaient par nature idéalement implémentables en Prolog. Cela explique en grande partie les nombreuses facilités de développement mentionnées. Mais il ne faut pas oublier la différence d'approche dans la résolution du problème, c'est-à-dire l'idée de spécifier un plan de solution et non de coder une solution particulière.

Cette apologie de l'approche logique en programmation doit cependant être modérée en fonction de l'objectif poursuivi. En effet, si notre approche s'est avérée plus adéquate au problème que celle choisie avant nous par Monsieur Bolle, cette adéquation peut être remise en question lors du développement de l'outil de production correspondant.

Les limites de notre prototype, qui ont été soulevées au chapitre 2, avaient souvent une solution rapide en Prolog, ou du moins pouvaient être réglées en faisant appel aux nouveaux concepts apportés par Prolog II ou par les logiques à sémantique enrichie. Les limites que nous y avons décrites ne concernaient cependant que le prototype lui-même.

Si le prototype se transforme en outil de production, il serait absurde de garder une programmation réalisée purement en Prolog, vu la lenteur bien connue de ce dernier. Cependant, la programmation des bases de connaissances et du traitement des commandes seront avantageusement conservées en Prolog. Toute l'interface opérateur,

la gestion d'erreurs et l'accès aux 'dumps' peuvent être implémentés dans un langage de programmation classique, ce choix améliorant les performances et la maîtrise du programme car Prolog est un langage encore très peu utilisé de nos jours.

Nous désirons montrer ainsi le danger de céder à l'apparente facilité du développement en Prolog. Bien que le code soit restreint et rapidement exécutable, la maîtrise de Prolog est nettement plus difficile que celle d'un langage procédural habituel. De plus, aucun support de génie logiciel n'existe pour Prolog.

Nous conseillons donc ici de ne garder Prolog que là où son aspect déclaratif est indispensable et d'utiliser un langage classique dès qu'une programmation procédurale est nécessaire, même si Prolog est parfaitement capable de l'assumer aussi.

Outre la programmation logique, notre application a également utilisé des bases de connaissances en lieu et place de bases de données. Si ce choix ne se justifie certainement pas par des critères de performance, nous le justifions par des motifs de maintenabilité et de développement incrémental. En effet, même si le modèle de données que nous avons utilisé est de type relationnel, nous y avons adjoint des connaissances au sens propre, donc qui dépassent le pouvoir d'expression des schémas relationnels. Un exemple en est la définition d'un pointeur pointé comme une table.

Ce genre de connaissances sera indispensable dans un outil de production car la philosophie des structures de données que nous devions décrire ne peut être considérée comme parfaitement homogène dans tout le logiciel. Des connaissances spécifiques à certaines parties de BS2000 devront certainement être introduites dans l'outil; la notion de monde de Prolog II, entre autre, se révélerait fort utile à cette fin.

Bien entendu, rien n'empêche l'outil de production de simuler la gestion de faits de Prolog, partie la plus importante des bases de connaissances, par une base de données, cela pour des raisons à nouveau de performances. Ce type de choix nous apparaît cependant, de par sa complexité, relever plus d'un choix stratégique quant à l'importance de l'outil à construire. Pour cette raison, il dépasse le cadre de ce travail.

Bibliographie.

[Amble 87] Tore Amble, *Logic Programming and Knowledge Engineering*, Addison Wesley Publishing Company, 1987.

[Bratko 86] Ivan Bratko, *Prolog programming for artificial intelligence*, International Computer Science Series, 1986.

[Clocksin 87] William F. Clocksin, Christopher S. Mellish, *Programming in Prolog*, Springer Verlag, troisième édition, 1987.

[Conlon 85] T. Conlon, *Start problem solving with Prolog*, Addison wesley Publishers, 1985.

[Dedié 88] Dr Günter Dedié, *Nucleus BS2000*, Siemens Aktiengesellschaft, München, 1988.

[Garrico 89] M. A. Garrico, J. E. Girard, J. P. Jones, *Building Knowledge Systems*, Intertext Publications, McGraw-Hill Book Company, 1989.

[GKPC 86] Françis Giannesini, Henry Kanoui, Robert Pasero, Michel van caneghem, *Prolog*, Addison Wesley Publishing Company, 1986.

[Hepburn 87] P. H. Hepburn, *Further programming in Prolog : writing application programs*, Ellis Horwood Limited, John Wiley & Sons, 1987.

[Kluzniak 85] Feliks Kluzniak, Stanislaw Szpakowicz, *Prolog for Programmers*, Academic Press, 1985.

[Lazarev 88] G. L. Lazarev, *Why Prolog ?*, Prentice Hall, New Jersey, 1988.

[Li 84] Deyi Li, *A Prolog database system*, Research Studies Press ltd, John Wiley & Sons Inc, 1984.

[Lucas 88] R. Lucas, *Database applications using Prolog*, Ellis Horwood limited, John Wiley & Sons, 1988.

[Maier] David Maier, *Chapter 3 : databases in the fifth generation project : is Prolog a database language ?*

[Naish] Lee Naish, *Concurrent database updates in Prolog*

[Shapiro 86] Leon Sterling, Ehud Shapiro, *The Art of Prolog*, The MIT Press, 1986.

[Thayse 88] André Thayse, *From standard logic to logic programming*, John Wiley & Sons, 1988.

[Turner 84] Raymond Turner, *Logics for Artificial Intelligence*, Ellis Horwood Limited, John Wiley & Sons, 1984.

[Walker 87] Adrian Walker, *Knowledge Systems and Prolog*, Addison Wesley Publishing Company, 1987.

Annexe

Cette annexe comporte le code Prolog de notre application : AIDA. Outre ce code, nous avons également créé une interface en C vers des procédures Pascal implémentant le protocole d'accès aux 'dumps' appelé Anita. Cette interface, très courte, a été testée mais n'a pu être intégrée à l'interpréteur Prolog que nous utilisions vu l'impossibilité d'obtenir tout le système de développement du Prolog sous BS2000. N'ayant pu effectuer cette intégration et n'étant donc pas sûr de ce travail, nous ne le présenterons pas ici. Le lecteur intéressé pourra contacter Monsieur Bolle, chez Siemens Rhisnes, pour de plus amples détails.

Le code Prolog de AIDA

Implémentation d'un langage de description des structures internes du système d'exploitation BS2000 de Siemens et d'interrogation de dumps, sous forme d'un système à bases de connaissances.

Application realisée par Ph.MOISSE d'octobre à décembre 1989.

Base de faits de test sur les tables du task management.

```
table(xvt).
contient(xvt,exvtlta,2892,4).
pointeur(exvtlta,etlt,0,1,1).
table(etlt).
```

```
contient(etlt,tltvect,40,32*4).
vecteur(tltvect,tltindx,4,3).                    % normalement 32 et pas 3
pointeur(tltindx,etcb,0,1,0).
table(etcb).
contient(etcb,test,0,4).                         %élément purement de test
contient(etcb,etcbtft,572,4).
pointeur(etcbtft,idtft,0,1,1).
table(idtft).
contient(idtft,idmp1fl,12,4).
pointeur(idmp1fl,id1fcb,0,1,1).
contient(idtft,idmp2fl,16,4).
pointeur(idmp2fl,id2fcb,0,1,1).
table(id1fcb).
contient(id1fcb,id1p2lnk,100,4).
pointeur(id1p2lnk,id2fcb,0,1,1).
table(id2fcb).
contient(id2fcb,id2tftlk,16,4).
pointeur(id2tftlk,idtft,0,1,1).
contient(id2fcb,id2p1lk,20,4).
pointeur(id2p1lk,id1fcb,0,1,1).
```

Base de faits sur les tables du memory management.

```
contient(evsmt,evcvect,evc3ppt,999).
vecteur(evcvect,evcppt,3,5).
pointeur(evcppt,ppta,0,1,1).
pointeur(ppta,pptqueue,0,1,1).
queue(pptqueue,ppt,pptql).
pointeur(pptql,ppt,0,1,1).
contient(ppt,pptvect,8,999).
vecteur(pptvect,ppte,1,0).
contient(ppte,pptevpt,0,999).
pointeur(pptevpt,ml,0,4096,ptevpl).
pointeur(ml,slqueue,ml,1,1).
```

```
queue(slqueue,sml,swd).
queue(bslqueue,sml,sbck).
pointeur(swd,sml,ml,1,0).
pointeur(sbck,sml,ml,1,0).
pointeursinverses(swd,sbck).
pointeur(ml,frqueue,ml,1,'ffwd<>0').
queue(frqueue,sml,fswd).
queue(bfrqueue,sml,fsbck).
pointeur(fswd,sml,sml,1,4080).
pointeur(fsbck,sml,sml,1,4080).
pointeursinverses(fswd,fsbck).
parcourir(sml,slqueue).
```

Programme principal : interface utilisateur de haut niveau.

```
run:-
    clearscreen,
    dessinerbord,
    messagedepresentation,
    interfaceutilisateur,
    findeprogramme.

dessinerbord:-
    ligne,
    nl,
    dessinercentre,
    ligne.

dessinercentre:-
    for(1,X,15),
    write('*                                                    '),
    writeln('                              *'),
    fail.
dessinercentre.
```

```
ligne:-
   write('**************************************************'),
   write('*************************').

messagedepresentation:-
   movecursor(2,14),
   write('AIDA : an Artificially Intelligent Dump Analyser.'),
   movecursor(5,24),
   write('Copyright (C) Ph. MOISSE 1990'),
   movecursor(9,37),
   write('|'),
   movecursor(10,36),
   write('-+-'),
   movecursor(11,37),
   write('|'),
   movecursor(14,21),
   write('Press Return key to begin execution.'),
   get0(X).

interfaceutilisateur:-
   clearscreen,
   ouvrirdump,
   suiteinterface.

suiteinterface:-
   writeln('Please enter your meta-command :'),
   compiler,
   writeln(''),
   writeln('Would you want to continue working with me ?'),
   lireligne(H),
   continuation(H).
suiteinterface:-
   writeln('The predicate COMPILER failed.'),
   suiteinterface.
```

```
continuation(H):-
    extraire(Mot,H,Hs),
    decompcons(Mot,Lettres),
    extraire(C,Lettres,Reste),
    veutcontinuer(C),
    !,
    clearscreen,
    suiteinterface.
continuation(C):-
    !.

veutcontinuer(C):-
    C == 'y';
    C == 'Y'.

findeprogramme:-
    fermerdump,
    clearscreen,
    dessinerbord,
    movecursor(6,28),
    write('See you soon, I hope !'),
    movecursor(14,21),
    write('Press Return key for the last time ...'),
    movecursor(15,36),
    get0(Key),
    clearscreen.
```

Conditions d'existence d'un chemin dans les données.

```
chemins(A,Z,Route):-
    adjacents(A,),
    adjacents(Z,),
    unchemin(A,[Z],Route).
```

```
unchemin(A,[A|Finderoute],[A|Finderoute]).
unchemin(A,[Y|Finderoute],Route):-
   adjacents(X,Y),
   not member(X,Finderoute),
   unchemin(A,[X,Y|Finderoute],Route).

adjacents(X,Y):-
   queue(X,Y,);
   contient(X,Y,,);
   vecteur(X,Y,,);
   pointeur(X,Y,,,).

presenterlechemin(X,Y):-
   chemins(X,Y,Res),
   presenterchemin(Res).

presenterchemin(Route):-
   extraire(Noeud1,Route,Routerestante),
   presenterchemin(Noeud1,Routerestante).
presenterchemin(X,[]).
presenterchemin(Noeud,Route):-
   extraire(Noeudsuivant,Route,Restederoute),
   presenterlarelationentre(Noeud,Noeudsuivant),
   presenterchemin(Noeudsuivant,Restederoute).

presenterlarelationentre(X,Y):-
   pointeur(X,Y,,,),write(X),write(' points on '),write(Y),nl;
   contient(X,Y,,),write(X),write(' contains '),write(Y),nl;
   vecteur(X,Y,,),write(X),write(' is a vector of '),write(Y),nl;
   queue(X,Y,),write(X),write(' is a '),doublequeue(X),
      write('queue of '),write(Y),nl.

doublequeue(Queue):-
   doublequeuedeux(Queue),
   write('double '),
   !.
```

```
doublequeue(Queue).
```

Utilitaire de saisie à l'écran des commandes de l'utilisateur.
(analyse lexicale)

```
lireligne(L):-
    get0(C),
    lireligne(C,L1),
    concatseparateurs(L1,L2),
    numeriserliste(L2,L).

lireligne(C,[L|Ls]):-
    caractere(C),
    liremot(C,L,C1),
    lireligne(C1,Ls).
lireligne(C,[L|Ls]):-
    separateur(C),
    charcode(L,C),
    get0(C1),
    lireligne(C1,Ls).
lireligne(C,Ls):-
    C == 64,
    get0(C1),
    lireligne(C1,Ls).
lireligne(C,[]):-
    C == 21.

liremot(C,W,C1):-
    caracteres(C,Cs,C1),
    name(W,Cs).
```

```
caracteres(C,[C|Cs],C0):-
   caractere(C),
   !,
   get0(C1),
   caracteres(C1,Cs,C0).
caracteres(C,[],C):-
   not caractere(C).
caractere(C):-129 =< C,C =< 137.                    % a ... i
caractere(C):-145 =< C,C =< 153.                    % j ... r
caractere(C):-162 =< C,C =< 169.                    % s ... z
caractere(C):-193 =< C,C =< 201.                    % A ... I
caractere(C):-209 =< C,C =< 217.                    % J ... R
caractere(C):-226 =< C,C =< 233.                    % S ... Z
caractere(C):-240 =< C,C =< 249.                    % 0 ... 9

separateur(187).                                    % [
separateur(189).                                    % ]
separateur(78).                                     % +
separateur(96).                                     % -
separateur(92).                                     % *
separateur(97).                                     % /
separateur(126).                                    % =
separateur(110).                                    % >
separateur(75).                                     % .
separateur(76).                                     % <
separateur(94).                                     % ;
separateur(107).                                    % ,

concatseparateurs([],[]).
concatseparateurs([H1|T1],[H1|T2]):-
   not separ(H1),
   concatseparateurs(T1,T2).
concatseparateurs([H1|T1],[H1|T2]):-
   separ(H1),
   extraire(Elt,T1,Tq),
   not separ(Elt),
```

```
    concatseparateurs(T1,T2).
concatseparateurs([H1|T1],[H2|T2]):-
    separ(H1),
    extraire(Elt,T1,Tq),
    separ(Elt),
    concat([H1,Elt],H2),
    concatseparateurs(Tq,T2).

separ(X):-
    atomiclength(X,Res),
    Res > 1,
    fail.
separ(X):-
    atomiclength(X,1),
    charcode(X,Res2),
    separateur(Res2),
    Res2 \== 187,                    % car [ et ] ne doivent pas être
    Res2 \== 189.                    % concaténés aux autres séparateurs

numeriser(X,Res):-
    name(X,L),
    number(Res,L).
numeriser(X,X).

numeriserliste([],[]).
numeriserliste([X|Y],[Z|YY]):-
    numeriser(X,Z),
    numeriserliste(Y,YY).
```

Définitions des relations de base.

```
table(X,Y,Z):-
    contient(X,Y,Z,).
```

```
table(X,X,0):-
   pointeur(X,,,,),
   pointeur(,X,,,).
      % car un pointeur pointé est considéré comme une table à un seul élément

doublequeuedeux(A):-
   queue(A,X,D),
   queue(B,X,E),
   pointeur(D,X,,,),
   pointeur(E,X,,,),
   ptrsinv(D,E).

ptrsinv(X,Y):-
   pointeur(X,Y,,,),
   pointeur(Y,X,,,).
ptrsinv(X,Y):-
   pointeursinverses(X,Y).
ptrsinv(X,Y):-
   pointeursinverses(Y,X).
```

Procédures d'exécution des instructions de l'utilisateur.
(analyse sémantique)

```
compiler:-
   asserta(erreurcompil(non)),
   lireligne(L),
   !,
   commande(L,[],cmd(Res1,Res2,Res3)),
   !,
   cmd(Res1,Res2,Res3).
compiler:-
   assertz(syntaxerror('End of error list.')),
   findall(Message,syntaxerror(Message),Lmes),
   writeln('I am sorry, but I found the following error :'),
```

```
    ecrireliste(Lmes),
    abolish(syntaxerror,1),
    abolish(erreurcompil,1),
    !,
    fail.

cmd(route(A),enregcond(B),exec(C)):-
    erreurcompil(X),
    !,
    X == non,
    abolish(erreurcompil,1),
    route(A),
    !,
    enregcond(B),
    !,
    exec(C).
```

Recherche des différentes routes possibles et permises.

```
route(X):-
    X =.. Y,
    lineariser(Y,Routedemandee),
    rajoutxvt(Routedemandee,Routedemandee2),
    extraire(Source,Routedemandee2,Lqcqe),
    extrairedernier(Dest,Routedemandee,Linconnue),
    suiteroute(Source,Dest,Routedemandee).

rajoutxvt(A,A):-
    extraire(Tete,A,C),
    Tete == 'xvt'.
rajoutxvt(A,B):-
    extraire('xvt',B,A).
```

```
suiteroute(S,D,Routedemandee):-
    chemins(S,D,Route),
    inclusion(Routedemandee,Route),
    asserta(path(Route)),
    write('Route : '),
    writeln(Route).
suiteroute(,,):-
    asserta(syntaxerror('Specified Path Unknown in Database.')),
    fail.

inclusion([],).
inclusion([X|Route1],Route2):-
    member(X,Route2),
    inclusion(Route1,Route2).
```

Complétion des conditions et création des références adresses des tables spécifiées.

```
enregcond(X):-
    X =.. Y,
    defairelcond(Y,Z),
    rajouteref(Z,W),
    asserta(listecond(W)),
    write('Liste de conditions : '),
    writeln(W).

defairelcond([lcond,[]],[]).
defairelcond([lcond,cond(A,B,C)],[cond(A,B,C)]).
defairelcond([lcond,cond(F,G,H)|X],[cond(F,G,H)|Y]):-
    extraire(B,X,Toto),
    B =.. A,
    defairelcond(A,Y).

rajouteref([],[]).
```

```
rajouteref([cond(X,Y,Z)|Lcond],[cond(X,Y2,Z2)|Lcond2]):-
    changer(Y,Y2),
    changer(Z,Z2),
    !,
    rajouteref(Lcond,Lcond2).

changer(X,ref(X,xvt)):-
    atomic(X),
    not numeric(X).
changer(X,X).

lineariser(X,Y):-
    destructurer(X,Z),
    clarifier(Z,Y).

destructurer(Xs,Ys):-
    destr(Xs,Ys-[]).

destr([],Xs-Xs).
destr(X,[X|Xs]-Xs):-
    cconst(X),
    X \== [].
destr([X|Xs],Ys-Zs):-
    destr(X,Ys-Ys1),
    destr(Xs,Ys1-Zs).

cconst(X):-
    atom(X);
    integer(X).
cconst(X):-
    functor(X,si2,3);
    functor(X,si1,2);
    functor(X,list,1);
    functor(X,display,1);
    functor(X,lcons,2);
    functor(X,cond,3);
```

```
    functor(X,atome,1).

clarifier([],[]).
clarifier([X|Xt],Yt):-
    indesirable(X),
    clarifier(Xt,Yt).
clarifier([X|Xt],[X|Yt]):-
    not indesirable(X),
    clarifier(Xt,Yt).

indesirable(X):-
    X == [];
    X == ';';
    X == '.'.
```

Recherche des occurences des tables spécifiées et lancement des instructions.

```
exec(X):-
    extraire(A,X,Toto),  % verifie si c'est une liste, comportant alors plusieurs
    extraire(B,Toto,Toto2),     %                                    instructions
    B \== [],
    X =.. Y,
    lineariser(Y,Z),
    write('Commandes : '),
    writeln(Z),
    lancergeneral(Z).
exec(X):-                              % ce n'est pas une liste, donc une seule instruction
    write('Commande : '),
    writeln(X),
    lancergeneral([X]).

lancergeneral(Linstr):-
    path(Chemin),
    listecond(Lcond),
```

```
    abolish(path,1),
    abolish(listecond,1),
    cstruiretable(Lcond),
    trouveroccurence(Chemin,Lcond,Linstr),
    abolish(reference,3),
    abolish(cond,3).
lancergeneral(Linstr):-
    writeln('No more occurrences.'),
    abolish(reference,3),
    abolish(cond,3).

cstruiretable([]).
cstruiretable([Cond|Lcond]):-
    asserta(Cond),
    cond(X,Y,Z),
    abolish(cond,3),
    ajouter(Y),
    ajouter(Z),
    !,
    cstruiretable(Lcond).

ajouter(ref(A,B)):-
    asserta(reference(A,B)).
ajouter(X):-
    numeric(X).                          % test logiquement superflu

trouveroccurence(Chemin,Lcond,Linstr):-
    extraire(Elt,Chemin,Chemin2),
    infodump(Xvtadresse,Erreur),
    !,
    parcourirchemin(Elt,Xvtadresse,Chemin2),
    adressefinale(X),
    abolish(adressefinale,1),
    testerconditions(Lcond),
    write('Occurence trouvee a l''adresse : '),
    writeln(X),
```

```
    lancer(Linstr).

parcourirchemin(,X,[]):-
    asserta(adressefinale(X)).
parcourirchemin(Elt,Adr,[Elt2|Chemin2]):-
    rempliradresses(Elt,Adr),
    veriflien(Elt,Elt2,Adr,Adr2),
    parcourirchemin(Elt2,Adr2,Chemin2).

veriflien(Elt1,Elt2,Adr1,Resultat):-
    contient(Elt1,Elt2,Depl,),
    Resultat is Adr1+Depl.
veriflien(Elt1,Elt2,Adr1,Resultat):-
    pointeur(Elt1,Elt2,,,Regle),
    verifreglepointeur(Regle),
    readdump(Adr1,Resultat,Erreur).
veriflien(Elt1,Elt2,Adr1,Resultat):-
    vecteur(Elt1,Elt2,Taillechacun,Nbre),
    for(0,X,Nbre-1),
    Toto is X*Taillechacun,
    Resultat is Adr1+Toto.

lancer([]).
lancer([Com|ListeCom]):-
    executer(Com),
    lancer(ListeCom).

verifreglepointeur().                          % non implémenté !

rempliradresses(Elt,Adr):-
    clause(reference(A,Elt),Body),
    !,
    retract(reference(A,Elt)),
    contient(Elt,A,Depl,),
    Adresse is Adr+Depl,
    asserta(reference(A,Elt,Adresse)),
```

```
    rempliradresses(Elt,Adr).
rempliradresses(,):-
    !.
```

Procédures de test des conditions générales sur les tables.

```
testerconditions([]).
testerconditions([Cond|Lcond]):-
    verifier(Cond),
    !,
    testerconditions(Lcond).

verifier(Cond):-
    arg(1,Cond,Oper),
    arg(2,Cond,Opd1),
    arg(3,Cond,Opd2),
    verifier2(Oper,Opd1,Opd2).
verifier2(Oper,Opd1,Opd2):-
    numeric(Opd1),
    numeric(Opd2),
    apploper(Oper,Opd1,Opd2).
verifier2(Oper,Opd1,Opd2):-
    numeric(Opd1),
    not numeric(Opd2),
    arg(1,Opd2,Elt2),
    arg(2,Opd2,Ref2),
    reference(Elt2,Ref2,Adr),
    readdump(Adr,Val,Erreur),
    apploper(Oper,Opd1,Val).
verifier2(Oper,Opd1,Opd2):-
    numeric(Opd2),
    not numeric(Opd1),
    arg(1,Opd1,Elt1),
    arg(2,Opd1,Ref1),
```

```
    reference(Elt1,Ref1,Adr),
    readdump(Adr,Val,Erreur),
    apploper(Oper,Val,Opd2).
verifier2(Oper,Opd1,Opd2):-
    not numeric(Opd1),
    not numeric(Opd2),
    arg(1,Opd1,Elt1),
    arg(2,Opd1,Ref1),
    reference(Elt1,Ref1,Adr1),
    readdump(Adr1,Val1,Erreur),
    arg(1,Opd2,Elt2),
    arg(2,Opd2,Ref2),
    reference(Elt2,Ref2,Adr2),
    readdump(Adr2,Val2,Erreur),
    apploper(Oper,Val1,Val2).

apploper(Oper,Val1,Val2):-
    write('Application de '),
    write(Oper),
    write(' a '),
    write(Val1),
    write(' et a '),
    write(Val2),
    writeln('.'),
    Appel =.. [Oper,Val1,Val2],
    call(Appel).

ppt(X,Y):-
    X < Y.

pgd(X,Y):-
    X > Y.

egal(X,Y):-
    X =:= Y.
```

```
notppt(X,Y):-
   not ppt(X,Y).

notpgd(X,Y):-
   not pgd(X,Y).

notegal(X,Y):-
   not egal(X,Y).
```

Exécution réelle des instructions de l'utilisateur.

```
executer(A):-
   write('Execution de : '),
   writeln(A),
   faire(A),
   get0(X).

faire(nothing).
faire(success).
faire(failure):-
   fail.
faire(dialog).                                    % à compléter
faire(display(X)):-
   X =.. Y,
   defairelcons(Y,Z),
   imprcontenu(Z).
faire(list(X)):-
   X =.. Y,
   defairelcons(Y,Z),
   imprcontenu2(Z).
faire(si1(X,Y)):-
   rajouteref([X],[Xref]),
   verifier(Xref),
   faire(Y).
```

```
faire(si2(X,Y,Z)):-
    rajouteref([X],[Xref]),
    verifier(Xref),
    faire(Y).
faire(si2(X,Y,Z)):-
    rajouteref([X],[Xref]),
    not verifier(Xref),
    faire(Z).

defairelcons([A],[A]).
defairelcons([lcons,A,B],[A,B]):-
    atomic(B).
defairelcons([lcons,A,lcons(W,X)],[A,W|Y]):-
    X =.. Z,
    defairelcons(Z,Y).

imprcontenu([]).
imprcontenu([X|Y]):-
    reference(X,,Adr),
    readdump(Adr,Val,Erreur),
    write(X),
    write(' = '),
    writeln(Val),
    imprcontenu(Y).

imprcontenu2([]).
imprcontenu2([X|Y]):-
    imprtous(X),
    imprcontenu2(Y).

imprtous(X):-
    reference(X,Y,Adr),
    readdump(Adr,Val,Erreur),
    write(X),
    write(' = '),
    writeln(Val),
```

```
    fail.
imprtous().
```

Analyseur syntaxique des instructions de l'utilisateur et base de connaissances sur le langage de commandes.
<commande> ::= 'with' <chemin> 'where' <liste cond> 'do' <liste instr> <chemin> ::= <atome> | <atome> '.' <chemin> <liste instr> ::= <instr> | <instr> ';' <liste instr> <instr> ::= <display> | <list> | 'dialog' | 'failure' | 'nothing' | 'if' <cond> 'then' <instr> 'else' <instr> | 'success' | 'if' <cond> 'then' <instr> <display> ::= 'display(' <liste const> ')' <list> ::= 'list(' <liste const> ')' <liste const> ::= <constante> | <constante> ',' <liste const> <liste cond> ::= <cond> | <cond> 'and' <liste cond> <cond> ::= <expression> <opcomp> <expression> <opcomp> ::= '<' | '>' | '=' | '<>' | '<=' | '>=' <constante> ::= atome prolog | <reference> <reference> ::= <constante> '[' <constante> ']' <expression> ::= <constante> <oparith> <expression> | <constante> <oparith> ::= '+' | '-' | '*' | '/'

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% commande(cmd(route(Path),enregcond(Cond),exec(Instr)))-->      %
%   [with],                                                      %
%   chemin(Path),                                                %
%   !,                                                           %
%   [where],                                                     %
%   listecond(Cond),                                             %
%   !,                                                           %
%   [do],                                                        %
%   listeinstr(Instr),                                           %
%   !.                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
commande([with|A],B,cmd(route(C),enregcond(D),exec(E))):-
   chemin(A,[where|F],C),
   !,
   listecond(F,[do|G],D),
   !,
   listeinstr(G,B,E),
   !.
commande(A,B,cmd(route([error]),enregcond([error]),exec([error]))):-
   asserta(erreurcompil(oui)),
   asserta(syntaxerror('With, Where or Do Missing.')).

%%%%%%%%%%%%%%%%%%%%
% chemin(T)-->                    %
%   eltchemin(T)                  %
% chemin([T1,T2])-->              %
%   eltchemin(T1),                %
%   ['.'],                        %
%   chemin(T2).                   %
%%%%%%%%%%%%%%%%%%%%

chemin(A,B,C):-
   eltchemin(A,B,C).
chemin(A,B,[C,D]):-
   eltchemin(A,['.'|E],C),
   chemin(E,B,D).
chemin(A,B,[error]):-
   asserta(erreurcompil(oui)),
   asserta(syntaxerror('Incorrect Path Specification.')).

%%%%%%%%%%%%%%%%%%%%%%
% eltchemin(X)-->[X].           %
%%%%%%%%%%%%%%%%%%%%%%

eltchemin([A|B],B,A).
```

```
%%%%%%%%%%%%%%%%%%
% listecond(lcond([]))-->[].      %
% listecond(lcond(L))-->          %
%   cond(L).                      %
% listecond(lcond(L1,L2))-->      %
%   cond(L1),                     %
%   [and],                        %
%   listecond(L2).                %
%%%%%%%%%%%%%%%%%%

listecond(A,A,lcond([])).
listecond(A,B,lcond(C)):-
   rcond(A,B,C).
listecond(A,B,lcond(C,D)):-
   rcond(A,[and|E],C),
   listecond(E,B,D).
listecond(A,B,[error]):-
   asserta(erreurcompil(oui)),
   asserta(syntaxerror('Incorrect Condition List.')).


%%%%%%%%%%%%%%%%%%
% instr(display(S))-->            %
%   [display],                    %
%   listeconst(S).                %
% instr(list(S))-->               %
%   [list],                       %
%   listeconst(S).                %
% instr(dialog)-->[dialog].       %
% instr(failure)-->[failure].     %
% instr(success)-->[success].     %
% instr(nothing)-->[nothing].     %
% instr(si2(T,S1,S2))-->          %
%   [if],                         %
%   cond(T),                      %
%   [then],                       %
```

```
%    instr(S1),                    %
%    [else],                       %
%    instr(S2).                    %
% instr(si1(T,S1))-->              %
%    [if],                         %
%    cond(T),                      %
%    [then],                       %
%    instr(S1).                    %
%%%%%%%%%%%%%%%%%%

instr([display|A],B,display(C)):-
    listeconst(A,B,C).
instr([list|A],B,list(C)):-
    listeconst(A,B,C).
instr([dialog|A],A,dialog).
instr([failure|A],A,failure).
instr([success|A],A,success).
instr([nothing|A],A,nothing).
instr([if|A],B,si2(C,D,E)):-
    rcond(A,[then|F],C),
    instr(F,[else|G],D),
    instr(G,B,E).
instr([if|A],B,si1(C,D)):-
    rcond(A,[then|E],C),
    instr(E,B,D).

%%%%%%%%%%%%%%%%%%
% listeinstr(S)-->                 %
%    instr(S).                     %
% listeinstr([S,Ss])-->            %
%    instr(S),                     %
%    [';'],                        %
%    listeinstr(Ss).               %
%%%%%%%%%%%%%%%%%%
```

```
listeinstr(A,B,C):-
   instr(A,B,C).
listeinstr(A,B,[C,D]):-
   instr(A,[;|E],C),
   listeinstr(E,B,D).
listeinstr(A,B,[error]):-
   asserta(erreurcompil(oui)),
   asserta(syntaxerror('Incorrect Instruction List')).

%%%%%%%%%%%%%%%%%%
% listeconst(S)-->constante(S).        %
% listeconst(lcons(S,Ss))-->     %
%   constante(S),                      %
%   [','],                             %
%   listeconst(Ss).                    %
%%%%%%%%%%%%%%%%%%

listeconst(A,B,C):-
   constante(A,B,C).
listeconst(A,B,lcons(C,D)):-
   constante(A,[','|E],C),
   listeconst(E,B,D).


%%%%%%%%%%%%%%%%%%
% expression(X)-->constante(X).        %
% expression(expr(Op,X,Y))-->          %
%   constante(X),                      %
%   oparith(Op),                       %
%   expression(Y).                     %
%%%%%%%%%%%%%%%%%%

expression(A,B,C):-
   constante(A,B,C).
expression(A,B,expr(C,D,E)):-
   constante(A,F,D),
```

```
    oparith(F,G,C),
    expression(G,B,E).

%%%%%%%%%%%%%%%%%%
% oparith('plus')-->['+'].          %
% oparith('moins')-->['-'].         %
% oparith('fois')-->['*'].          %
% oparith('divise')-->['/'].        %
%%%%%%%%%%%%%%%%%%

oparith([+|X],X,plus).
oparith([-|X],X,moins).
oparith([*|X],X,fois).
oparith([/|X],X,divise).

%%%%%%%%%%%%%%%%%%
% constanteunaire(X)-->[X].    %
%%%%%%%%%%%%%%%%%%

constanteunaire([A|B],B,A):-
    atomic(A).

%%%%%%%%%%%%%%%%%%
% constante(ref(X,Y))-->            %
%    constanteunaire(X),            %
%    ['['],                         %
%    constanteunaire(Y),            %
%    [']'].                         %
% constante(X)-->[X].               %
%%%%%%%%%%%%%%%%%%

constante(A,B,ref(C,D)):-
    constanteunaire(A,['['|E],C),
    constanteunaire(E,[']'|B],D).
constante([A|B],B,A):-
    atomic(A).
```

```
%%%%%%%%%%%%%%%%%%
% cond(cond(Op,X,Y))-->          %
%    expression(X),              %
%    opcomp(Op),                 %
%    expression(Y).              %
%%%%%%%%%%%%%%%%%%

rcond(A,B,cond(C,D,E)):-
    expression(A,F,D),
    opcomp(F,G,C),
    expression(G,B,E).

%%%%%%%%%%%%%%%%%%
% opcomp('egal')-->['='].        %
% opcomp('notegal')-->['<>'].    %
% opcomp('ppt')-->['<'].         %
% opcomp('pgd')-->['>'].         %
% opcomp('notpgd')-->['<='].     %
% opcomp('notppt')-->['>='].     %
%%%%%%%%%%%%%%%%%%

opcomp([=|X],X,egal).
opcomp([<>|X],X,'notegal').
opcomp([<|X],X,ppt).
opcomp([>|X],X,pgd).
opcomp([<=|X],X,'notpgd').
opcomp([>=|X],X,'notppt').
```

Interface réelle avec le dump physique via ANITA.
Procédures rajoutées au systeme pour l'appel d'Anita : - opendump(DumpName,Error) : ouvre le dump de nom specifié. - closedump(Error) : ferme le dump précédemment ouvert. - infodump(Address,Error) : fournit l'adresse de la table XVT. - readdump(Address,Data,Error) : lit le contenu du dump à l'adresse virtuelle fournie.
Lorsque, pour une raison quelconque, Anita renvoie une erreur, le prédicat Prolog correspondant ne fait qu'échouer, sans remonter le message d'erreur fourni par Anita.

```
ouvrirdump:-
   writeln('Please enter the dump file name to be opened :'),
   write('   '),
   lireligne(Dumpfilename),
   opendump(Dumpfilename,Error),
   writeln('Anita successfully opened your dump.').
ouvrirdump:-
   write('I got a problem : Anita cannot open your dump, please try'),
   writeln(' another.'),
   ouvrirdump.

fermerdump:-
   closedump().

opendump(,).

closedump().
infodump(0,0).

readdump(X,X,0).
```

Utilitaires divers.

```
writeln(X):-
    write(X),
    nl.

ecrireliste([]):-
    writeln('.').
ecrireliste([H|T]):-
    write(H),
    write(','),
    ecrireliste(T).

extrairedernier(L,[L],[]).
extrairedernier(X,[A|B],[A|Y]):-
    extrairedernier(X,B,Y).

extraire(X,[],[]).
extraire(X,[X|Y],Y).

ajoutfin([],X,X).
ajoutfin(X,[],[X]).
ajoutfin([X|T],Y,Res):-
    ajoutfin(X,Y,A),
    ajoutfin(T,A,Res).
ajoutfin(X,Y,Res):-
    ajoutdebut(X,[],A),
    ajoutdebut(Y,A,Res).

ajoutdebut([],X,X).
ajoutdebut(X,[],[X]).
ajoutdebut([X|T],Y,Res):-
    ajoutdebut(T,Y,A),
    ajoutdebut(X,A,Res).
ajoutdebut(X,Y,[X|Y]).
```

```
sauvertout:-
    tell('listingtotal.pro'),
    listing,
    told.

chargertout:-
    reconsult(listingtotal).

sauverbase:-
    tell('basedefaits.pro'),
    listing(contient),
    listing(pointeur),
    listing(pointeursinverses),
    listing(vecteur),
    listing(queue),
    listing(parcourir),
    told.

chargerbase:-
    reconsult(basedefaits).
```

Table des matières